TU WIEN Informatics

Automated Software Verification using Superposition-based Theorem Proving

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl. Ing. Bernhard Gleiss, BSc
Registration Number 00904987

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. techn. Laura Kovács, MSc
Second advisor: Univ. Prof. Dr. Matteo Maffei

The dissertation has been reviewed by:

Forename Surname	Forename Surname

Vienna, 15th September, 2020

Bernhard Gleiss

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

Erklärung zur Verfassung der Arbeit

Dipl. Ing. Bernhard Gleiss, 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, 15. September 2020

Bernhard Gleiss

Abstract

This thesis explores the automated verification of software programs involving loops and arrays using superposition-based theorem proving. It proposes new techniques at the intersection of program semantics, software verification, and automated reasoning.

In the first part of the thesis, we introduce an expressive instance of first-order logic modulo theories, called trace logic. We present a sound and complete axiomatic semantics of software programs and use it to capture the partial correctness of program properties as validity statements in trace logic. Our semantics describes each timepoint in the execution of a program uniquely and keeps program locations explicit. We then introduce a verification framework, which handles the inductive reasoning needed in the trace logic domain in order to enable off-the-shelf first-order theorem provers to reason about validity statements in trace logic. Our framework adds instances of an expressive induction axiom scheme explicitly into the search space, in contrast to other approaches which use induction axioms only on the meta-level. We then discuss how the combination of explicit induction axioms and backward-reasoning enables automated loop splitting, which is the key for verifying advanced array-properties. We conclude the first part of the thesis by generalizing the trace-logic-based verification framework to support multiple execution traces and relational properties.

In the second part of the thesis, we apply superposition-based theorem proving, and in particular the state-of-the-art theorem prover Vampire, to reason about the validity statements of the trace logic domain. We introduce two techniques, layered clause selection and subsumption demodulation, which are crucial for efficient superposition-based reasoning in the trace logic domain. Moreover, to ease the manual analysis of both proofs and failed proof attempts, we describe a tool, which interactively visualizes proof attempts of saturation-based theorem provers.

Finally, we provide an experimental evaluation of the trace-logic-based verification framework on interesting sets of benchmarks, which cover challenging properties of software programs including loops and arrays. The presented results suggest that our superposition-based verification framework is an interesting alternative to existing SMT-based verification approaches.

Kurzfassung

Diese Arbeit untersucht die automatisierte Überprüfung von Computerprogrammen, welche Arrays und Schleifen beinhalten, durch auf Superposition basiertes Theorembeweisen. Sie beschreibt neue Techniken am Schnittpunkt der Forschungsfelder Semantik von Programmiersprachen, Überprüfung von Computerprogrammen, und Automatisiertes Theorembeweisen.

Im ersten Teil der Arbeit beschreiben wir Trace Logic, eine ausdrucksstarke Instanz der durch Theorien erweiterten Prädikatenlogik erster Stufe. Wir präsentieren eine korrekte und vollständige axiomatische Semantik von Computerprogrammen, und verwenden diese, um die partielle Korrektheit von Programmeigenschaften als Gültigkeit logischer Formeln zu charakterisieren. Unsere Semantik beschreibt jeden Zeitpunkt der Ausführung eines Computerprogrammes eindeutig und erhält gleichzeitig die Programmstruktur. Aufbauend auf Trace Logic führen wir dann einen neuen Ansatz zur Überprüfung von Computerprogrammen ein. Dieser stellt das in Trace Logic benötigte induktive Schlussfolgern zur Verfügung, und ermöglicht in Folge das automatisierte Beweisen von Gültigkeitsaussagen in Trace Logic durch beliebige existierende Theorembeweiser für Prädikatenlogik. Unser Ansatz benutzt Instanzen eines ausdruckstarken Schemas von Induktionsaxiomen explizit im Schlussfolgern, im Gegensatz zu anderen Ansätzen, welche Induktionsaxiome nur auf der Metaebene verwenden. Wir erörtern dann, wie die Kombination von expliziten Induktionsaxiomen und rückwärts-gesteuertem Schlussfolgern das automatische Zerteilen von Schleifen in einfachere Teile simuliert und damit das Überprüfen anspruchvoller Eigenschaften von Array-Programmen ermöglicht. Als Abschluss des ersten Teils dieser Arbeit verallgemeinern wir unseren Ansatz, sodass mehrere Programmausführungen dargestellt und relationale Eigenschaften überprüft werden können.

Im zweiten Teil dieser Arbeit wenden wir das auf Superposition basierte Theorembeweisen, und insbesondere den führenden Theorembeweiser Vampire, an, um Gültigkeitsaussagen in Trace Logic zu beweisen. Wir beschreiben zwei Techniken, Layered Clause Selection und Subsumption Demodulation, die unverzichtbar sind für das effiziente Schlussfolgern in Trace Logic mittels Superposition. Um die manuelle Analyse von Beweisen und fehlgeschlagenen Beweisversuchen zu vereinfachen, beschreiben wir weiters ein Tool zur interaktiven Analyse von Beweisversuchen von auf Saturation basierten Theorembeweisern. Am Ende dieser Arbeit präsentieren wir die Ergebnisse einer experimentellen Auswertung unseres Überprüfungsansatzes auf interessanten Benchmarks,

die anspruchsvolle Eigenschaften von Computerprogrammen mit Arrays und Schleifen beinhalten. Die präsentierten Resultate legen nahe, dass unser auf Superposition basierter Überprüfungsansatz eine interessante Alternative zu existierenden SMT-basierten Ansätzen darstellt.

Acknowledgements

Foremost, I want to thank my thesis advisor Laura Kovács for her guidance, support, and encouragement throughout the journey of my PhD. Laura was always available if I needed advice and guided me when necessary, but gave me at the same time the space to grow as an independent researcher. Moreover, she connected me with many interesting people from TU Wien and abroad, which lead to inspiring meetings and productive collaborations.

Then, I want to thank my co-advisor Matteo Maffei, whose lecture on Formal Methods for Security and Privacy initiated my interest in reasoning about security properties. I highly enjoyed the stimulating whiteboard group discussions on applying Vampire in various security domains, which often went on for several hours.

Next, I want to thank Stephan Schulz and Philipp Rümmer, who kindly agreed to review this thesis. I understand that reviewing a thesis takes a substantial amount of time and should not be taken for granted.

This thesis would not have been possible without the famous Martin Suda, who always took the time to answer my endless stream of questions about superposition-based reasoning and Vampire. I am grateful for the invaluable explanations and the exciting collaborations later on in my PhD. Moreover, his insistence on well-versed prose helped me to improve my writing style considerably. I also want to thank Max Jaroschek, whose constructive feedback was very valuable throughout my PhD. Furthermore, he helped me to navigate through the early academic life and reminded Martin and me occasionally that PDR is nothing to discuss over lunch.

I want to especially thank Arie Gurfinkel, who is one of the kindest researchers I got to know in my PhD. During my research stay with him, I not only learned a lot about program verification and research in general, but also improved substantially as a software engineer. Arie furthermore made my research stay a pleasant experience, and introduced me to DVLB, the best coffeehouse in Waterloo, which developed for the time of my research stay into my main workspace. I would also like to thank Rodrigo and Thorsten, for showing me the campus life at UWaterloo, and for the many memorable moments during that time.

Many PhD students contributed to the friendly work environment at TU Wien and made the countless meetings, sessions, and coffee breaks both entertaining and worthwhile. For

this, I want in particular thank Andi, Pamina, Jakob, and Renate.

This PhD would not have been possible without the many friends, who provided a relaxing and motivating environment outside of work as well as new perspectives on the sometimes challenging academic life when they became necessary. Thank you Flo, Martina, Jlu, Rolando, Conni, Maggi, Kadie, Raoul, Julian, Thomas, Phöl, Reinhold, Mors, Armin, Nina, Julia, Jakob, and Anna, for all the joyful moments and amusing conversations that I would not want to miss.

Another important factor for finishing this thesis was my family, which always encouraged me on my way. I want to thank Karl and Roswitha, who are the supporting and caring parents everyone wishes for, and Michi and Wolfi, who are the best brothers I can imagine, for being there for me at all time.

Finally, I'm grateful that I could experience my PhD with a very special person at my side. Thank you Chri, for being part of this journey, and for the wonderful time we are spending together.

This work was funded by the ERC Starting Grant 2014 SYMCAR 639270, the ERC Proof of Concept Grant 2018 SYMELS 842066, the Wallenberg Academy Fellowship 2014 TheProSE, the Austrian FWF research projects W1255-N23 and RiSE S11409-N23, and the OMAA Grant 101öu8.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Background

A need for reliable software. Our society depends on critical infrastructure, which is automatically controlled by software. Examples of such infrastructure are (i) traffic lights, (ii) medical devices like infusion pumps or X-ray machines, (iii) airplanes and cars controlled by autopilots, (iv) power grids, (v) digital financial services, ranging from traditional banking applications to distributed ledgers, and (vi) storage services for user data, both on-premise and in the cloud, which are used by individuals, companies or governments.

In each of these examples, the controlling software needs to fulfill certain properties. For instance, (i) traffic lights should not signal at the same time to two participants on conflicting lanes to enter an intersection, (ii) medical devices must avoid that a patient is harmed by excessive doses of medication or radiation, (iii) autopilots should neither crash the vehicle they are navigating nor crash themselves during navigation, (iv) power grids need to ensure that global power is continuously supplied even in the presence of local power outages, (v) digital financial services need to correctly track who owns what, and (vi) user data has to be kept private from unauthorized access. If the software is not implemented correctly, such properties can be violated, which can cause devastating consequences: In the case of traffic lights, medical devices, and autopilots, human life may be at risk. For power grids, other systems like water supply, communication, and public transportation could be heavily affected. In the case of digital financial services, huge financial losses may occur, and for data breaches, a multitude of financial, legal, and social consequences could arise. It should be noted that the violation of a property due to an incorrect implementation may either occur by coincidence or be forced by a malicious attacker.

The standard approach to decrease the chance of implementation errors in a software program is to perform *software testing*, that is, to execute the program on some inputs /

in some environment and check that the given properties are fulfilled for these executions. The advantage of this approach is that we do not need to know at all how the execution proceeds (we treat the program as a black-box), as we only need to observe the effects of executing the program. But testing also has the big disadvantage that we can only check a finite number of executions – and usually only a small amount of the overall executions. We claim that if a software is used to control critical infrastructure, it is not enough to only use testing to ensure that important properties of the software are satisfied. As a strong argument supporting this claim, all the properties discussed above have been violated in one or multiple real-world system(s) in the past due to incorrect software implementations, even though presumably these systems have been extensively tested.

A different approach, referred to as *software verification*, does not suffer from the mentioned shortcoming of testing. It ensures the absence of implementation errors in a software program by reasoning about *how* the program is executed, with the goal of concluding that the given properties are fulfilled for *all* inputs/environments. We can see that humans already verify software in their heads, as part of designing and implementing programs. Each programmer has developed a (custom) mental model of how programs are executed, which he/she uses to conclude that a given program fulfills the desired properties. Unfortunately, such human reasoning is often imprecise and sometimes flawed, as reasoning about executions of programs is both tedious and error-prone.

To overcome the problems of ad hoc and imprecise software verification, we can resort to a refined and more disciplined approach which we refer to as *formal software verification*. It is centered around a *precise and unambiguous language*, in which we both formulate a description of how the execution of the program is executed (this description is referred to as the *semantics of the program*), and the property whose correctness we want to establish. We then derive a detailed step-by-step argument in that language, which starts from the semantics of the program, ends in the given property, and explains why any execution of the given program has to fulfill that property (such an argument is referred to as a *proof of the property*). Which language should we choose to do all that in? We do not want to use natural language, as it would be hard then to be unambiguous and concise at the same time. Instead, we choose one of the formal languages based on (first-order) logic. Such languages loosely correspond to very restricted versions of natural language and have been specifically designed to be unambiguous, precise, and concise. Moreover, we know how to write compelling and understandable proofs in these languages. Finally, we can even automatically check proofs formulated in such languages with a computer, to ensure that these proofs do not contain any human mistakes.

Automated software verification. Formal software verification as described so far involves the problem that coming up with a proof is labor-intense and tedious. As a computer can check whether a manually generated proof contains no mistakes, we could wonder, whether the computer can automatically generate the proof itself in the first place. This idea leads to so-called *automated software verification*. With such an approach in

place, a programmer is only left with writing the program and specifying the properties he/she wants the program to fulfill, and can then ask a computer to automatically produce proofs, which show that the given properties hold and that no errors have been introduced. Unfortunately, the automated generation of proofs for program properties is a difficult task. A vast amount of research has been performed in the last 50 years, centering around the questions which semantics and which automated reasoning algorithm to use in the context of software verification [DKW08, HH19, ABB+16, CJGK+18, CC14]. It should be stressed that these two questions are closely related. A more expressive semantics can capture more knowledge about program executions, but can make the required reasoning more difficult. On the other hand, a more powerful reasoning algorithm can handle more expressive semantics, and in turn lead to a more powerful approach.

In recent years, *SMT-solving* [NOT06, Seb07, BT18] has developed into a mature reasoning technique for first-order logic, and has arguably become the most prominent technology for the automated generation of proofs for software verification. Verification frameworks based on either Hoare logic semantics [Hoa69] or Horn clause semantics [BGMR15] utilize the efficiency of SMT-solvers [DMB08, BCD+11], resulting in previously unseen automation for software verification, which also scales well in many cases. As a consequence, SMT-based software verification is increasingly used in industry, in particular for software controlling critical infrastructure [Coo18, BBDL+17].

Superposition-based theorem proving [BG94, NR01] is another technique for reasoning about problems formulated in first-order logic. While both SMT-solving and superposition-based theorem proving represent state-of-the-art reasoning techniques for first-order logic [Sut16, WCD+19], they focus on different areas. SMT-solvers excel at reasoning with quantifier-free problems, and provide advanced support for difficult background theories. While these solvers have been extended with support for quantification [DMB07, RTDM14, Rey16, RBF18], it is still challenging for them to perform efficient and stable reasoning under the presence of non-trivial quantification [LM09, Mos09, Rot16, Sut16]. Current superposition-based provers cannot compete with SMT-solvers on quantifier-free problems, in particular, if complicated theory reasoning is required [WCD+19]. On the other hand, these provers are natively designed with quantification in mind and provide efficient reasoning with problems containing *arbitrary quantification*. It has been conjectured that a superposition-based software verification approach could be an interesting alternative to existing SMT-based approaches, as the efficient handling of quantification inside superposition-based theorem provers could be used to support programs and properties, for which existing approaches do not provide stable and efficient reasoning yet. In particular, efficient support for quantification could enable (i) new formalizations for abstract program features like unbounded arrays, data structures, or function calls, and (ii) improved reasoning about program properties, which already contain quantification explicitly.

Surprisingly, there has not been much work exploring applications of saturation-based theorem proving to software verification, in particular to standard while-like programs. To apply superposition-based reasoners, it would make sense to formulate the semantics

of programs using standard semantics of first-order logic, in a way such that the (partial) correctness of a property is captured directly as a validity statement in standard first-order logic modulo suitable theories. Interestingly, for while-like programs, no such formalization has been described in the literature yet. We are therefore interested in the following research questions:

- Is it possible to formalize the semantics of software programs including loops and arrays directly in standard first-order logic modulo a suitable background theory? Can we use quantification to obtain a more expressive semantics?
- Is it possible to use superposition-based theorem proving to reason about software program properties? Can we use it to establish the partial correctness of programs containing loops and arrays which cannot be handled by current state-of-the-art approaches? Should superposition-based software verification be considered as an alternative to SMT-based approaches?

1.2 Contributions

This thesis provides the following main contributions:

- We introduce *trace logic*, a new instance of standard first-order logic modulo difference logic and integer arithmetic, and formulate an axiomatic semantics of software programs in it. Our axiomatic semantics preserves the structure of the program, does not abstract away intermediate timepoints in the execution, and avoids using any intermediate program logic. We furthermore establish the soundness and completeness (relative to Hoare logic) of the introduced semantics (Chapter 2).
- We present a new *software verification framework* based on trace logic, which can leverage any off-the-shelf theorem prover for first-order logic modulo difference logic and integer arithmetic to reason about program properties formulated in trace logic. To provide such a prover with the capabilities for inductive reasoning and automated loop splitting, we identify and describe a set of lemmas covering general inductive consequences, which are useful to establish the partial correctness of a wide range of software program properties (Chapter 3).
- We generalize the trace-logic-based software verification framework to *multiple executions and hyperproperties*, to automatically establish the partial correctness of interesting relational properties coming from security applications (Chapter 4).
- We investigate *in-depth* how to apply superposition-based theorem proving for reasoning in the trace-logic domain. In particular, we introduce new techniques, including *a layered clause selection heuristics* and a new simplification inference rule called *subsumption demodulation*, to speed up superposition-based theorem proving on the trace logic domain (Chapters 6 and 7). Moreover, we present the tool SatVis, which enables the efficient analysis of proofs and saturation attempts of the saturation-based first-order theorem prover Vampire (Chapter 8).
- We implemented the ideas discussed in this thesis in the tools Rapid and Vampire and experimentally evaluate our approach on interesting sets of benchmarks. The

results suggest that our approach is a promising alternative to existing approaches (Chapter 9).

1.3 Publications

This thesis is based on the following publications.

[GGK20] Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace logic for inductive loop reasoning. In *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design (FMCAD 2020)*, pages 255–263. TU Wien Academic Press, 2020

[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

[GKR20] Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption demodulation in first-order theorem proving. In *Proceedings of the 10th International Joint Conference on Automated Reasoning (IJCAR 2020)*, volume 12166 of *LNCS*, pages 297–315. Springer, 2020

[GS20b] Bernhard Gleiss and Martin Suda. Layered clause selection for theory reasoning (short paper). In *Proceedings of the 10th International Joint Conference on Automated Reasoning (IJCAR 2020)*, volume 12166 of *LNCS*, pages 402–409. Springer, 2020

[GS20a] Bernhard Gleiss and Martin Suda. Layered clause selection for saturation-based theorem proving. In *Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR 2020)*. Accepted for Publication

[GKS19] Bernhard Gleiss, Laura Kovács, and Lena Schnedlitz. Interactive visualization of saturation attempts in Vampire. In *Proceedings of the 15th International Conference on Integrated Formal Methods (IFM 2019)*, volume 11918 of *LNCS*, pages 504–513. Springer, 2019

Furthermore, the following publications are an additional result of the PhD leading up to this thesis.

[GKS17] Bernhard Gleiss, Laura Kovács, and Martin Suda. Splitting proofs for interpolation. In *Proceedings of the 26th International Conference on Automated Deduction (CADE 2017)*, volume 10395 of *LNCS*, pages 291–309. Springer, 2017

[SG18] Martin Suda and Bernhard Gleiss. Local soundness for QBF calculi. In *Proceedings of the 21st International Conference on Theory and Applications of*

Satisfiability Testing (SAT 2018), volume 10929 of *LNCS*, pages 217–234. Springer, 2018

[GKR18] Bernhard Gleiss, Laura Kovács, and Simon Robillard. Loop analysis by quantification over iterations. In *Proceedings of the 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2018)*, volume 57 of *EPiC Series in Computing*, pages 381–399. EasyChair, 2018

1.4 Outline

This thesis is organized as follows.

In the first part, we focus on formalizing the partial correctness of properties of software programs using first-order logic. Chapter 2, which has already occurred in simpler form in [BEG+19] and [GGK20], develops a new semantics to formalize the partial correctness of programs, using a novel instance of first-order logic modulo theories called trace logic. Chapter 3 shows how to build a verification framework using the semantics from Chapter 2 in combination with an arbitrary theorem prover supporting first-order logic. It is based on [GGK20], except for Sections 3.1 and 3.3, which have not been published yet. Chapter 4, which is based on [BEG+19], generalizes the ideas from Chapters 2 and 3 to multiple computation traces and hyperproperties.

In the second part, we focus on applying a superposition-based theorem prover to reason about validity statements generated by the ideas of Chapters 3 and 4. Chapter 5 recalls the relevant background on superposition-based theorem proving and discusses how to tune the superposition-based theorem prover Vampire to the trace-logic domain. Both Chapter 6, based on [GS20] and [GS], and Chapter 7, based on [GKR20], develop custom reasoning techniques to speed up superposition-based theorem proving on the trace logic domain. Chapter 8, which is based on [GKS19], investigates how to analyze proof attempts in order to get new insights into how to optimize both the encoding of program correctness and superposition-based reasoning in the trace-logic domain.

Finally, Chapter 9 presents an experimental evaluation of the ideas of this thesis, followed by a conclusion and future work in Chapter 10. Preliminary versions of the experimental results have already been reported in [BEG+19] and [GGK20].

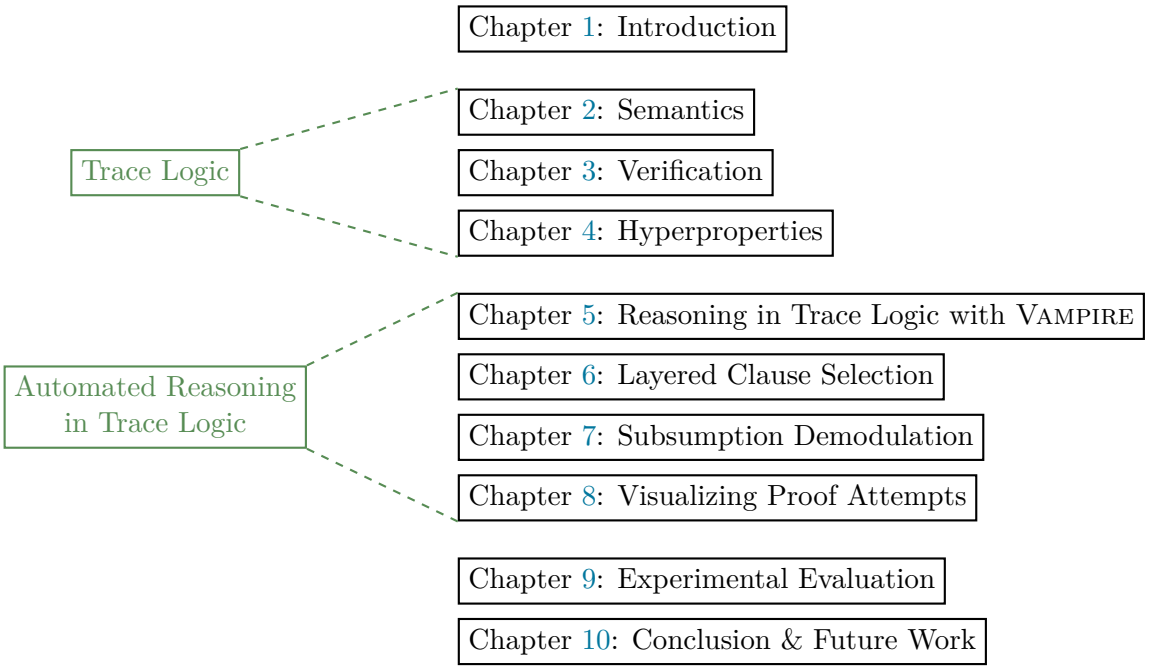

Figure 1.1: Structure of the thesis.

CHAPTER 2

Trace Logic and Semantics

In this chapter, we present a novel logical approach to formalize the semantics of imperative programs containing loops and arrays, geared towards the automated verification of functional properties. Our approach formalizes the execution of a program as well as its functional properties in *trace logic*, an expressive instance of many-sorted first-order logic with equality. Trace logic draws its expressiveness from its syntax, which allows expressing properties over computation traces. It supports fine-grained reasoning about intermediate steps in program executions, using explicit loop iterations.

The main advantages of trace logic over existing approaches are:

- Using trace logic, we can *uniquely describe each timepoint* in the execution of a given program (including loops) *with a finite language*, while at the same time keeping program locations explicit.
- Trace logic allows arbitrary quantification over iterations and values of program variables. In particular, we can express and reason with (i) generalized induction axioms, usable to simulate advanced loop splitting, and (ii) iterations that depend on (possibly non-ground) expressions involving program variables.
- With trace logic, we formalize the semantics of a program directly as axioms in *standard first-order logic, using standard first-order semantics*. In particular, any execution of a program corresponds to a valid interpretation of our axiomatization. Note that this is only possible since trace logic can express each timepoint of the execution uniquely. Our direct encoding into first-order logic has several advantages: (i) we avoid the use of an intermediate program logic, which would complicate our framework, (ii) we inherit the monotonicity of entailment[1], a main requirement for modular reasoning, and (iii) we can directly apply general techniques of first-order logic, such as automatic reasoning, proof theory, and interpolation.

[1] A logical sytem is called monotone with respect to entailment, if any property provable from some axioms A remains provable if we extend A with additional axioms.

```
 1  func main()
 2  {
 3     const Int[] a;
 4     const Int alength;
 5     Int[] b;
 6     Int blength = 0;
 7
 8     Int i = 0;
 9     while(i < alength)
10     {
11        if (a[i] >= 0)
12        {
13           b[blength] = a[i];
14           blength = blength + 1;
15        }
16        else
17        {
18           skip;
19        }
20        i = i + 1;
21     }
22  }
23
```

Figure 2.1: Running example.

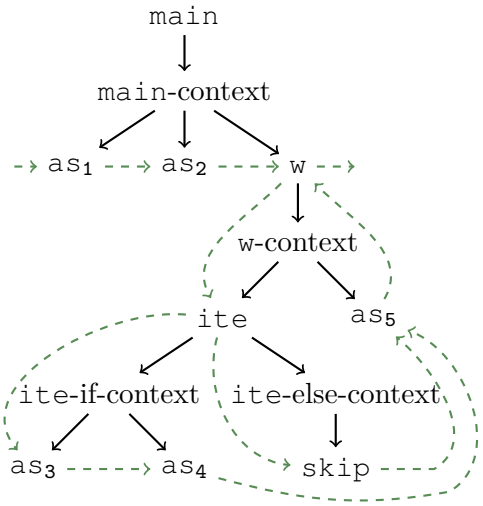

Figure 2.2: Program tree of running example.

We motivate our work with the simple program of Figure 2.1. This program iterates over an integer-valued array `a` and copies each positive element into a new array `b`. Our aim is to prove the following property: At the end of the execution of the program, for any position *bpos* in `b`, there exists a position *apos* in `a`, such that the element in `a` at position *apos* is equal to the element in in `b` at position *bpos*. We formalize this property as

$$\forall bpos^{\mathbb{I}}.(0 \leq bpos < blength \rightarrow \exists apos^{\mathbb{I}}.a(apos) \simeq b(end, bpos)), \tag{2.1}$$

where $apos^{\mathbb{I}}$ and $bpos^{\mathbb{I}}$ respectively specify that *apos* and *bpos* are of sort integer $\mathbb{I}$. Further, $a(apos)$ denotes the value of the element at position *apos* of `a`, whereas *end* refers to the last program location of Figure 2.1 (that is, line 23).

In the remaining part of this chapter, we will develop axiomatic semantics, which let us prove the correctness of property (2.1) in standard first-order logic modulo difference logic and integer arithmetic.

2.1 Preliminaries

2.1.1 First-Order Logic

We consider standard many-sorted first-order logic with equality, where equality is denoted by $\simeq$. We allow all standard boolean connectives and quantifiers in the language and write $s \not\simeq t$ instead of $\neg(s \simeq t)$, for two arbitrary first-order terms s and t. A *signature* is any finite set of symbols. We consider equality $\simeq$ as part of the language; hence, $\simeq$ is not a symbol. Given a logical variable x and sort S, we write x^S to denote that the sort of x is S.

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.

By a *(first-order) background theory*, or simply just *theory*, we mean the set of all formulas valid on a class of first-order structures. When we discuss a theory, we call symbols occurring in the signature of the theory *interpreted*, and all other symbols *uninterpreted*. In our work, we consider the combination (union) $\mathbb{D} \cup \mathbb{I}$ of the theory $\mathbb{D}$ of difference logic over natural numbers and the one $\mathbb{I}$ of integers. The signature of $\mathbb{D}$ consists of standard function- and predicate symbols `0`, `succ`, `pred`, and $<$, respectively interpreted as *zero*, *successor*, *predecessor*, and *less*. Note that $\mathbb{D}$ does not contain interpreted symbols for (arbitrary) addition and multiplication. We use the theory $\mathbb{D}$ to represent and reason about loop iterations (see Section 2.2). The signature of $\mathbb{I}$ consists of the standard integer constants $0, 1, 2, \ldots$, function symbols $+$ and $*$, and the predicate symbol $<$. We use the theory $\mathbb{I}$ to represent and reason about integer-valued program variables (see Section 2.2). Additionally, we use two (uninterpreted) sorts as two sets of uninterpreted symbols: (i) the sort Timepoint, written as $\mathbb{L}$, for denoting timepoints in the execution of a program, and (ii) the sort Trace, written as $\mathbb{T}$, for denoting computation traces of a program.

We use standard first-order interpretations/models modulo a theory T, and in particular modulo $\mathbb{D} \cup \mathbb{I}$. We write $\vDash_T F$ to denote that F holds in all models of T (and hence is valid). If I is a model of T, we write $I \vDash_T F$ if F holds in the interpretation I.

2.1.2 Induction

Recall the *standard induction axiom scheme* for (difference logic over) natural numbers. For any first-order formula IH with a free variable of sort $\mathbb{D}$ and no other free variables, the scheme contains the induction axiom

$$\Big(BC \wedge IC\Big) \rightarrow Concl,$$

where BC, IC, and $Concl$ are macros (the so-called *base case*, *inductive case*, resp. *conclusion*), which are defined as follows:

$$BC := IH(\mathtt{0}) \tag{2.2}$$
$$IC := \forall it^{\mathbb{D}}.(IH(it) \to IH(\mathtt{succ}(it))) \tag{2.3}$$
$$Concl := \forall it^{\mathbb{D}}.IH(it) \tag{2.4}$$

Instead of this standard axiom scheme, we will use throughout this thesis the so-called *bounded induction axiom scheme*, which allows us to reason inductively about a *bounded* interval $[it_L, it_R]$. For any first-order formula IH with a free variable of sort $\mathbb{D}$ and no other free variables, and for any terms it_L and it_R of sort $\mathbb{D}$, the scheme contains the induction axiom

$$\Big(BC(it_L) \wedge IC(it_L, it_R)\Big) \to Concl(it_L, it_R),$$

where BC, IC, and $Concl$ are defined as follows:

$$BC(it_L) := IH(it_L) \tag{2.5}$$
$$IC(it_L, it_R) := \forall it^{\mathbb{D}}.\Big((it_L \leq it < it_R \wedge IH(it)) \to IH(\mathtt{succ}(it))\Big) \tag{2.6}$$
$$Concl(it_L, it_R) := \forall it^{\mathbb{D}}.\Big(it_L \leq it \leq it_R \to IH(it)\Big) \tag{2.7}$$

We can see that the bounded induction axiom scheme contains arbitrary (fixed) left and right bounds, in contrast to the standard induction axiom scheme, which fixes the left bound to $\mathtt{0}$ and does not include a right bound. Note that an instance of the bounded induction axiom scheme is neither entailed by nor entails the corresponding instance of the standard induction axiom scheme, as the former has both a logically weaker premise and a logically weaker conclusion than the latter.

We will additionally also use a generalized version of the bounded induction axiom scheme, the so-called *generalized bounded induction axiom scheme*, which allows universal quantification over both the interval bounds, which are used in the induction axioms, and the free variables of the induction hypothesis.

For any first-order formula IH with a free variable of sort $\mathbb{D}$ and free variables $x_1 \ldots, x_k$ of sort $\mathbb{I}$ (where k is allowed to be 0), the scheme contains the induction axiom

$$\forall x_1^{\mathbb{I}}, \ldots, x_k^{\mathbb{I}}, it_L^{\mathbb{D}}, it_R^{\mathbb{D}}.\Bigg(\Big(BC(it_L) \wedge IC(it_L, it_R)\Big) \to Concl(it_L, it_R)\Bigg),$$

where BC, IC, and $Concl$ are defined as follows:

$$BC(it_L) := IH(it_L) \tag{2.8}$$
$$IC(it_L, it_R) := \forall it^{\mathbb{D}}.\Big((it_L \leq it < it_R \wedge IH(it)) \to IH(\mathtt{succ}(it))\Big) \tag{2.9}$$
$$Concl(it_L, it_R) := \forall it^{\mathbb{D}}.\Big(it_L \leq it \leq it_R \to IH(it)\Big) \tag{2.10}$$

program ::= `func main() { context }`
context ::= statement; ... ; statement
statement ::= var := expr
| array-var[expr] := expr
| `skip`
| `if ( expr ) { context } else { context }`
| `while ( expr ) { context }`

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

2.1.3 Programming Model $\mathcal{W}$

We consider programs written in a standard while-like programming language, denoted as $\mathcal{W}$, including arrays and arbitrary nestings of if-then-else- and while-statements.

More precisely, the language is defined as follows: $\mathcal{W}$ includes mutable and constant integer- and integer-array-variables, integer constants, and standard side-effect-free expressions `+`, `-`, `*`, `==`, `<`, `!`, `&&` and `||` over integers and booleans. On top of these variables and expressions, each program in $\mathcal{W}$ consists of a single top-level function `main` and arbitrarily nested statements, as defined by the grammar in Figure 2.3. Note that each statement in Figure 2.3 is either an assignment, array-assignment, skip-statement, if-then-else-statement, or while-statement. Throughout this thesis, whenever we refer to *loops*, we mean while-statements. A *subprogram* is either a context or a statement.

We use contexts to capture lists of statements. We furthermore use subprograms to capture parts of the program, which have a start- and an end-point of execution, even though they might not be programs themselves (as they lack a top-level function statement).

Throughout the thesis, we will only consider *terminating* programs. A generalization of our approach to non-terminating programs is out of the scope of this thesis, but should be possible (for instance, by extending the ideas presented in [GKR18]).

2.1.4 Program Trees

Throughout this chapter, the so-called *program trees* will be central for describing the execution of a program. Intuitively, a program tree corresponds to the directed syntax tree of the program, where expressions and variables are ignored.

2.1. Definition Let $\mathtt{p}_0$ be a program. The *program tree of* $\mathtt{p}_0$ is the directed tree T, such that (i) the root of T is the (single) top-level function statement of $\mathtt{p}_0$, (ii) the top-level function statement `func main(){ c }` of $\mathtt{p}_0$ and any while-statement `while(expr){c}` of $\mathtt{p}_0$ have as only child-node the context `c`, (iii) any if-then-else-

statement `if(expr){c1} else {c2}` has as only child-nodes the contexts c_1 and c_2, and (iv) any context $s_1;\ldots;s_k$ has as only child-nodes the statements $s_1, \ldots, s_k$.

The *top-level context of* p_0 is the (single) child-node of the top-level function statement in the program tree. A *top-level statement of* p_0 is any child-node of the top-level context. For a context $c := s_1;\ldots;s_k$, we (i) say that s_j *follows* s_i for any $1 \leq i < j \leq k$, (ii) say that s_i *occurs* in c for any $1 \leq i \leq k$, and (iii) define the *last statement* as s_k. For any subprogram p, the set of *enclosing loops* consists of all while-statements w, such that w is a transitive parent of p in the program tree.

2.2. Example Consider the program p_0 of the running example from Figure 2.1. Let w denote the loop starting at line 9, let ite denote the if-then-else-statement starting at line 11, and let $as_1, \ldots, as_5$ denote the assignments at lines 6, 8, 13, 14, resp. 20. The program tree of p_0 is visualized in Figure 2.2 (where solid lines denote the edges in the tree). The top-level statements are as_1, as_2, and w. Assignment as_4 follows as_3, and both as_3 and as_4 occur in the context of ite. Assignment as_4 is the last statement of the if-context, and the only enclosing loop of as_4 is w. □

We conceptually take a static outside-view of the execution of the program, where the execution proceeds on the program tree, by moving a so-called program-pointer around the statements in the program tree, without changing or rewriting the program tree itself. In each step, the program-pointer is moved to a different statement and the values of program variables are updated. Finally, the program-pointer is moved to a distinguished end-location in the program, and the execution of the program stops. In the next two sections, we will make this conceptual idea precise.

Program trees may appear similar to transition systems (where states denote locations and edges denote transitions from one location to another while changing values of program variables), since both of them allow to describe the execution of the program as transitions on a static structure. But compared to transition systems, program trees additionally keep the nesting of statements and loops explicit. In contrast to other semantics, this information is important for us, as we will use it in later definitions, in particular in Subsection 2.2.1 and Subsection 2.4.1.

2.3. Example Consider again the program tree from Figure 2.2. The possible transitions of the program pointer between statements are visualized using dashed arrows. □

2.2 Trace Logic

In this section, we introduce a language to describe how a program is executed by moving the program-counter around the program tree as explained in Subsection 2.1.4.

2.2.1 Locations and Timepoints

For each program statement s, we introduce a program location l_s. We denote by l_{end} the location corresponding to the end of the program.

As program locations can be revisited during program executions due to the presence of loops, we model locations as follows. For each location l_s corresponding to a program statement $\mathtt{s}$, we introduce a function symbol l_s with target sort $\mathbb{L}$ in our language, denoting the timepoint where the interpreter visits the location. For each enclosing loop of the statement $\mathtt{s}$, the function symbol l_s has an argument of type $\mathbb{D}$; this way, we distinguish between different iterations of the enclosing loop of $\mathtt{s}$. We denote the set of all such function-symbols l_s as S_{Tp}. When $\mathtt{s}$ is a loop, we additionally include a function symbol n_s with target sort $\mathbb{D}$ and an argument of sort $\mathbb{D}$ for each enclosing loop of $\mathtt{s}$. This way, n_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 such function symbols n_s as S_n.

2.4. Example Consider again Figure 2.1. We abbreviate each statement $\mathtt{s}$ by the line number of the first line of $\mathtt{s}$. We use l_6 to refer to the timepoint corresponding to the first assignment of $\mathtt{blength}$ in the program. We denote by $l_9(\mathtt{0})$ and $l_9(n_9)$ the timepoints corresponding to evaluating the loop condition in the first and, respectively, last loop iteration. Further, we write $l_{11}(it)$ and $l_{11}(\mathtt{succ}(\mathtt{0}))$ for the timepoint corresponding to the beginning of the loop body in the it-th and, respectively, second iteration of the loop. Note that $\mathtt{succ}(\mathtt{0})$ is a term algebra expression of $\mathbb{D}$. □

For simplicity, let us define macros over the most commonly used timepoints. First, define it^s to be a function, which returns for each while-statement $\mathtt{s}$ a unique variable of sort $\mathbb{D}$. We use this function to consistently name variables denoting loop iterations. Second, let $\mathtt{s}$ be a statement, let $\mathtt{w_1}, \ldots, \mathtt{w_k}$ be the enclosing loops of $\mathtt{s}$ and let it be an arbitrary term of sort $\mathbb{D}$.

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

Third, let $\mathtt{p}$ be an arbitrary subprogram, that is, let $\mathtt{p}$ be either a statement or a context. We refer to the timepoint where the execution of $\mathtt{p}$ has started (parameterized by the enclosing iterators) by

$$
start_{\mathtt{p}} := \begin{cases} tp_{\mathtt{p}}(\mathtt{0}) & \text{if } \mathtt{p} \text{ is while-statement} \\ tp_{\mathtt{p}} & \text{if } \mathtt{p} \text{ is non-while statement} \\ start_{\mathtt{s_1}} & \text{if } \mathtt{p} \text{ is context } \mathtt{s_1};\ldots;\mathtt{s_k} \end{cases}
$$

Fourth, for an arbitrary subprogram $\mathtt{p}$, let $end_{\mathtt{p}}$ denote the timepoint which follows immediately after $\mathtt{p}$ has been evaluated completely (including the evaluation of subpro-

grams of p):

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

Fifth, let s be the first statement of the top-level context, and define

$$start := start_{\mathtt{s}}.$$

Finally, let s be a statement with enclosing loops $\mathtt{w_1}, \ldots, \mathtt{w_k}$. We will use $\forall enclIts_s.\ F$ to denote the formula $\forall it^{w_1}, \ldots, it^{w_k}.\ F$, for an arbitrary trace logic formula F.

2.2.2 Reachability

We introduce a predicate $Reach : \mathbb{L} \mapsto \mathbb{B}$, which intuitively captures the set of timepoints reached in an execution. Note that in our work, reachability is defined as a predicate over timepoints, in contrast to defining reachability as a predicate over program-configurations. We thereby decouple the reachability of a timepoint tp from the values of the program variables at that timepoint tp. We use S_R to denote the set $\{Reach\}$.

2.2.3 Program Variables and Expressions

In our setting, we reason about program behavior by expressing properties over program variables v. To do so, we capture the value of program variables v at timepoints (from $\mathbb{L}$) . Hence, we model program variables v as functions $v : \mathbb{L} \mapsto \mathbb{I}$, where $v(tp)$ gives the value of v at timepoint tp. If the program variable v is an array, we add an additional argument of sort $\mathbb{I}$, which corresponds to the position at which the array is accessed. We denote by S_V the set of such introduced function symbols denoting program variables. We finally model arithmetic constants and program expressions using integer functions.

Note that our setting can be simplified for non-mutable variables – in this case, we omit the timepoint argument in the function representation of the variable.

2.5. Example Consider again Figure 2.1. By $blength(l_8)$ we refer to the value of program variable blength at the moment before i is first assigned. We write $b(l_{11}(it), i(l_{11}(it)))$ for the value of array b at timepoint $l_{11}(it)$ at position pos, where pos is the value of i at timepoint $l_{11}(it)$. As a is unchanged in the program, we write $a(i(l_{11}(it)))$ for the value of array a at position pos. We denote by $i(l_{end}(it)) + 1$ the value of the expression i+1 at the end of the execution. □

From now on, for an arbitrary program expression e, define $[\![\mathtt{e}]\!](tp)$ to denote the value of e at timepoint tp.

2.2.4 Trace Logic $\mathcal{L}$

We now have all ingredients to define our *trace logic* $\mathcal{L}$.

The signature of $\mathcal{L}$ contains the symbols of the theories $\mathbb{D}$ and $\mathbb{I}$ together with the symbols introduced in Subsection 2.2.1-2.2.3, that is, symbols denoting timepoints, last iterations in loops, the reachability of timepoints, and program variables.

Formally,

$$Sig(\mathcal{L}) \;:=\; (S_{\mathbb{D}} \cup S_{\mathbb{I}}) \;\cup\; (S_{Tp} \cup S_n \cup S_R \cup S_V).$$

We define *trace logic*, denoted by $\mathcal{L}$, as the instance of many-sorted first-order logic with equality modulo $\mathbb{D} \cup \mathbb{I}$ with signature $Sig(\mathcal{L})$.

2.2.5 Definitions of Commonly used Formulas

We finally introduce definitions to express how the values of program variables evolve between two timepoints. These definitions will be used heavily while defining the semantics in Section 2.3 and Section 2.4.

Consider $v \in S_V$, that is a function denoting a program variable $\mathtt{v}$, let $\mathtt{e}, \mathtt{e}_1$, and $\mathtt{e}_2$ be program-expressions, and let tp_1, tp_2 denote two timepoints. First define:

$$Eq(v, tp_1, tp_2) := \begin{cases} \forall pos^{\mathbb{I}}.\ v(tp_1, pos) \simeq v(tp_2, pos), & \text{if } \mathtt{v} \text{ is array} \\ v(tp_1) \simeq v(tp_2), & \text{otherwise} \end{cases} \tag{2.11}$$

That is, $Eq(v, tp_1, tp_2)$ in (2.11) states that the program variable $\mathtt{v}$ has the same values at tp_1 and tp_2. Second, define

$$EqAll(tp_1, tp_2) := \bigwedge_{v \in S_V} Eq(v, tp_1, tp_2), \tag{2.12}$$

asserting that all program variables have the same values at the timepoints tp_1 and tp_2. Third, define

$$Update(v, e, tp_1, tp_2) := v(tp_2) \simeq [\![\mathtt{e}]\!](tp_1) \wedge \bigwedge_{v' \in S_V \setminus \{v\}} Eq(v', tp_1, tp_2), \tag{2.13}$$

Fourth, define

$$\begin{aligned} UpdateArr(v, e_1, e_2, tp_1, tp_2) := \quad & \forall pos^{\mathbb{I}}.(pos \not\simeq [\![\mathtt{e}_1]\!](tp_1) \rightarrow \\ & v(tp_2, pos) \simeq v(tp_1, pos)) && \text{(2.14a)} \\ \wedge \quad & v(tp_2, [\![\mathtt{e}_1]\!](tp_1)) \simeq [\![\mathtt{e}_2]\!](tp_1) && \text{(2.14b)} \\ \wedge \quad & \bigwedge_{v' \in S_V \setminus \{v\}} Eq(v', tp_1, tp_2) && \text{(2.14c)} \end{aligned}$$

2.3 Small-step Operational Semantics

In this section, we present standard small-step operational semantics of $\mathcal{W}$ [Plo04], rephrased in trace logic. We will use these small-step operational semantics in Section 2.4 for establishing the soundness of the axiomatic semantics (also introduced in Section 2.4).

Our presentation is semantically equivalent to standard small-step operational semantics [Plo04], but differs syntactically in several points, to simplify later definitions and theorems: (i) we annotate while-statements with counters to ensure the uniqueness of timepoints during the execution, (ii) we reference nodes in the program-tree to keep track of the current location during the execution instead of using strings to denote the remaining program (iii) we avoid additional constructs like states or configurations (iv) we keep the timepoints in the execution separated from the values of the program variables at these timepoints, and (v) we evaluate expressions on the fly.

2.3.1 Transition Rules

First, we formalize single steps of the execution of the program as transition rules, as defined in Figure 2.4. Intuitively, the rules describe (i) how we move the location-pointer around on the program-tree and (ii) how the state changes while moving the location-pointer around. Each rule consists of (i) a premise $\mathit{Reach}(tp_1)$ for some timepoint tp_1, (ii) an additional premise F (omitted if F is $\top$), the so-called *side-condition*, which is an arbitrary trace-logic formula referencing only the timepoint tp_1, (iii) the first conjunct of the conclusion of the form $\mathit{Reach}(tp_2)$ for some timepoint tp_2, and (iv) the second conjunct of the conclusion, which again is an arbitrary trace-logic formula G referencing only the timepoints tp_1 and tp_2. Note that each rule which is applied to a statement `s` contains a free variable it^w for each enclosing loop `w` of `s`. A *transition rule instance* is the result of substituting each such variable with a concrete integer value.

2.3.2 Execution Interpretations and Partial Correctness

Next, we now formalize the possible executions of the program as a set of first-order interpretations, the so-called *execution interpretations*.

In a nutshell, execution interpretations can be described as follows. Each possible execution of the program induces an interpretation. For each such execution, the predicate symbol *Reach* is interpreted as the set of timepoints reached during the execution. The function symbols denoting values of program variables are interpreted according to the transition rules at the timepoints reached during the execution, and are interpreted arbitrarily at all other timepoints.[2]

We construct execution interpretation iteratively, as follows: We move around the program as defined by the transition rules. Whenever we reach a new timepoint, we choose

[2]This is the standard way to encode partial functions in first-order logic.

$$[init^{sos}]\ \frac{}{Reach(start)}$$

Let s be a `skip`.

$$[skip^{sos}]\ \frac{Reach(start_{\mathtt{s}})}{Reach(end_{\mathtt{s}}) \wedge EqAll(start_{\mathtt{s}}, end_{\mathtt{s}})}$$

Let s be an assignment `v = e`.

$$[asg^{sos}]\ \frac{Reach(start_{\mathtt{s}})}{Reach(end_{\mathtt{s}}) \wedge Update(v, e, start_{\mathtt{s}}, end_{\mathtt{s}})}$$

Let s be an array-assignment `v[e1] = e2`.

$$[asg^{sos}_{arr}]\ \frac{Reach(start_{\mathtt{s}})}{Reach(end_{\mathtt{s}}) \wedge UpdateArr(v, e_1, e_2, start_{\mathtt{s}}, end_{\mathtt{s}})}$$

Let `s` be `if(Cond){c1}else{c2}`.

$$[ite^{sos}_{T}]\ \frac{Reach(start_{\mathtt{s}}) \qquad [\![\mathtt{Cond}]\!](start_{\mathtt{s}})}{Reach(start_{\mathtt{c}_1}) \wedge EqAll(start_{\mathtt{s}}, start_{\mathtt{c}_1})}$$

$$[ite^{sos}_{F}]\ \frac{Reach(start_{\mathtt{s}}) \qquad \neg[\![\mathtt{Cond}]\!](start_{\mathtt{s}})}{Reach(start_{\mathtt{c}_2}) \wedge EqAll(start_{\mathtt{s}}, start_{\mathtt{c}_2})}$$

Let `s` be `while(Cond){c}`.

$$[while^{sos}_{T}]\ \frac{Reach(tp_{\mathtt{s}}(it^s)) \qquad [\![\mathtt{Cond}]\!](tp_{\mathtt{s}}(it^s))}{Reach(start_{\mathtt{c}}) \wedge EqAll(tp_{\mathtt{s}}(it^s), start_{\mathtt{c}})}$$

$$[while^{sos}_{F}]\ \frac{Reach(tp_{\mathtt{s}}(it^s)) \qquad \neg[\![\mathtt{Cond}]\!](tp_{\mathtt{s}}(it^s))}{Reach(end_{\mathtt{s}}) \wedge EqAll(tp_{\mathtt{s}}(it^s), end_{\mathtt{s}})}$$

Figure 2.4: Small-step operational semantics using *tp*, *start*, *end*.

a program state J', such that the side-conditions of the transition rule are fulfilled, and extend the current interpretation J with J'. We furthermore collect all timepoints that we already reached in I. We stop as soon as we reach *end*. We then construct an execution interpretation as follows: we interpret *Reach* as I, extend J to an interpretation of S_V by choosing an arbitrary state at any timepoint which we did not reach, and choose an arbitrary interpretation of the theory symbols according to the background theory.

2.6. Definition (Program state) A *program state at timepoint tp* is a partial interpretation, which exactly contains (i) for each non-array variable $\mathtt{v}$ an interpretation of $v(tp)$ and (ii)) for each array variable $\mathtt{a}$ and for each element *pos* of the domain $S_{\mathbb{I}}$ an interpretation of $a(tp, pos)$.

2.7. Definition (Execution-interpretation) Let $\mathtt{p_0}$ be a fixed program. Let I, J be any possible result returned by the algorithm in Algorithm 1. Let M be any interpretation, such that 1) $Reach(tp)$ is true iff $tp \in I$, 2) M is an extension of J, and 3) M interprets the symbols of the background theory according to the theory. Then M is called an *execution interpretation of* $\mathtt{p_0}$.

Algorithm 1 Algorithm to compute execution interpretation.

$J =$ choose program state at *start*
$I = \{start\}$
$curr = start$
while $curr \neq end$ **do**
  choose a transition rule instance $r := \dfrac{\sigma Reach(curr) \qquad \sigma F}{\sigma Reach(next) \wedge \sigma G}$, such that $J \vDash \sigma F$
  if r is $[while_F^{sos}]$ for some statement $\mathtt{s}$ **then**
    $J = J \cup \{\sigma lastIt_s \mapsto \sigma it^s\}$
  choose a program state J' such that $J \cup J' \vDash \sigma G$.
  $J = J \cup J'$
  $I = I \cup \{next\}$
  $curr = next$
return I, J

With the definition of execution interpretations at hand, we now define the (partially) correct properties of a program as the properties, which hold in each execution interpretation.

2.8. Definition Let $\mathtt{p_0}$ be a fixed program. Let F be a trace logic formula. Then F is called *partially correct* with respect to $\mathtt{p_0}$, if F holds in each execution interpretation of $\mathtt{p_0}$.

2.3.3 Simple Properties of Execution Interpretations

We conclude this subsection by stating simple properties of execution interpretations, which will be used in Section 2.5 and Section 2.6. The first property states that whenever

we reach the start of the execution of a subprogram p, we also reach the end of the execution of p. The second property states that whenever we reach the start of the execution of a context c, we also reach the start of the execution of each statement occurring in c. The third property states that whenever we reach the start of the execution of a while-statement s, then (i) we also reach the loop-condition check of s in each iteration up to and including the last iteration, and (ii) we also reach the start of the execution of the context of the loop body of s in each iteration before the last iteration.

2.9. Lemma Let $\mathtt{p}_0$ be a fixed program and let M be an execution interpretation of $\mathtt{p}_0$. Let further p be an arbitrary subprogram of $\mathtt{p}_0$ and σ be an arbitrary grounding of the enclosing iterations of p such that $\sigma Reach(start_{\mathtt{p}})$ holds. Then:

1. $\sigma Reach(end_{\mathtt{p}})$ holds in M.
2. If p is a context, $\sigma Reach(start_{\mathtt{s_i}})$ holds in M for any statement $\mathtt{s_i}$ occurring in p.
3. If p is a while-statement `while(Cond){c}`, then
   a. $\sigma Reach(tp_{\mathtt{p}}(it^p))$ holds in M for any iteration $it^p \leq \sigma(lastIt_{\mathtt{p}})$.
   b. $\sigma\sigma' Reach(start_c)$ holds in M for any iteration with $\sigma' it^p < \sigma(lastIt_{\mathtt{p}})$, where σ' is any grounding of it^p.

We prove all three properties using a single induction proof.

Proof. We proceed by structural induction over the program structure with the induction hypothesis

$$\forall enclIts_p.(Reach(start_{\mathtt{p}}) \rightarrow Reach(end_{\mathtt{p}})).$$

Let p now be an arbitrary subprogram of $\mathtt{p}_0$. For an arbitrary grounding σ of the enclosing iterations assume that $\sigma Reach(start_{\mathtt{p}})$ holds in M. To show that $\sigma Reach(end_{\mathtt{p}})$ holds in M, we perform a case distinction on the type of p:

- Assume p is **`skip`**, or an integer- or array-assignment: Since $\sigma Reach(start_{\mathtt{p}})$ holds in M, the rule $skip^{sos}$ resp. asg^{sos} resp. asg^{sos}_{arr} applies, so $\sigma Reach(end_{\mathtt{p}})$ holds in M too.
- Assume p is a context $\mathtt{s_1};\ldots;\mathtt{s_k}$. By definition, we know $start_p = start_{s_1}$, therefore $\sigma Reach(start_{\mathtt{s_1}})$ holds in M. By the induction hypothesis, we know that $\sigma Reach(start_{\mathtt{s_i}}) \rightarrow \sigma Reach(end_{\mathtt{s_i}})$ holds in M for any $1 \leq i \leq k$. Using a trivial induction, we conclude that $\sigma Reach(end_{\mathtt{s_i}})$ holds in M for any $1 \leq i \leq k$.
- Assume p is **`if`**`(Cond){c1}` **`else`** `{c2}`. Assume w.l.o.g. that $\sigma[\![\mathtt{Cond}]\!](start_{\mathtt{p}})$ holds in M. Then the rule ite^{sos}_T applies, so $\sigma Reach(start_{\mathtt{c_1}})$ holds in M. Using the induction hypothesis, we get $\sigma Reach(start_{\mathtt{c_1}}) \rightarrow \sigma Reach(end_{\mathtt{c_1}})$, so $\sigma Reach(end_{\mathtt{c_1}})$ holds in M. By definition, $end_c = end_{\mathtt{p}}$, so $\sigma Reach(end_{\mathtt{p}})$ holds in M.
- Assume p is **`while`**`(Cond){c}`. We perform a bounded subinduction over it^p from 0 to $\sigma lastIt_{\mathtt{p}}$ with the induction hypothesis $\sigma Reach(tp_{\mathtt{p}}(it^p))$.
  The base case holds, since $\sigma Reach(start_{\mathtt{p}})$ is the same as $\sigma Reach(tp_{\mathtt{p}}(0))$.

For the inductive case, assume that both $\sigma\sigma' Reach(tp_{\mathrm{p}}(it^p))$ and $\sigma'(it^p) < \sigma(lastIt_{\mathrm{p}})$ holds for some grounding σ' of it^p with the goal of deriving $\sigma\sigma' Reach(tp_{\mathrm{p}}(\mathtt{succ}(it^p)))$. Then rule $while_T^{sos}$ applies, so $\sigma\sigma' Reach(start_c)$ holds in M. From the induction hypothesis, we conclude $\sigma\sigma' Reach(start_c) \rightarrow \sigma\sigma' Reach(end_c)$, so $\sigma\sigma' Reach(end_c)$ holds. By definition, we know that $end_c = tp_p(\mathtt{succ}(it^p))$ holds in M, so we conclude that $\sigma\sigma' Reach(tp_p(\mathtt{succ}(it^p)))$ holds in M.
We have established the base case and the inductive case, so we apply bounded induction to derive that

$$\forall it^p.(\sigma(it^p) \leq \sigma(lastIt_{\mathrm{p}}) \rightarrow \sigma Reach(tp_{\mathrm{p}}(it^p)))$$

holds in M. In particular, $\sigma Reach(lastIt_{\mathrm{p}})$ holds in M. Since, by definition, also $\sigma\neg[\![\mathtt{Cond}]\!](lastIt_{\mathrm{p}})$ holds in M, we deduce that $while_F^{sos}$ applies, so $\sigma Reach(end_{\mathrm{p}})$ holds too.

□

2.4 Axiomatic Semantics

In this section, we state an axiomatic semantics of $\mathcal{W}$. We first define *Reach*, and then use *Reach* to define the semantics of an arbitrary program.

2.4.1 An Axiomatization of *Reach*

The following definition axiomatizes *Reach* using trace logic formulas. We will use *Reach* to define our semantics. Moreover, the predicate is useful for specifying properties about intermediate timepoints (since those properties can only hold if the referred timepoints are reached) and for reasoning about which locations are reached (which could be used in future work to reason about which functions are called in an execution).

2.10. Definition (*Reach*-predicate) For any context c, let

$$Reach(start_c) := \begin{cases} true & \text{if } \mathtt{c} \text{ is toplevel-context} \\ Reach(start_{\mathrm{s}}) \wedge [\![\mathtt{Cond}_s]\!](start_{\mathrm{s}}) & \text{if } \mathtt{c} \text{ context of if-branch of } \mathtt{s} \\ Reach(start_{\mathrm{s}}) \wedge \neg[\![\mathtt{Cond}_s]\!](start_{\mathrm{s}}) & \text{if } \mathtt{c} \text{ context of else-branch of } \mathtt{s} \\ Reach(start_{\mathrm{s}}) \wedge it^s < lastIt_{\mathrm{s}} & \text{if } \mathtt{c} \text{ context of body of } \mathtt{s}, \end{cases}$$

For any non-while statement $\mathtt{s}$ occurring in context $\mathtt{c}$, let

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

and for any while-statement s occurring in context $\mathtt{c}$, let

$$Reach(tp_{\mathrm{s}}(it^s)) := Reach(start_{\mathrm{c}}) \wedge it^s \leq lastIt_{\mathrm{s}}.$$

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

2.4.2 Non-Recursive Axiomatic Semantics using *Reach*

We now define the axiomatic semantics. Our semantics are not defined using recursion. We define axioms for each statement of the program and define the semantics of the program as the conjunction of all these axioms.

Let $\mathtt{p}_0$ be a fixed program. The semantics of $\mathtt{p}_0$ consists of a conjunction of one implication per statement, where each implication has the reachability of the start-timepoint of the statement as the premise and the semantics of the statement as the conclusion:

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

The semantics of the statements are defined as follows.

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

$$[\![\mathtt{s}]\!] := EqAll(end_{\mathtt{s}}, start_{\mathtt{s}}) \tag{2.15}$$

Integer assignments. Let $\mathtt{s}$ be an assignment $\mathtt{v = e}$, where $\mathtt{v}$ is an integer program variable and $\mathtt{e}$ is an expression. We reason as follows. The assignment $\mathtt{s}$ is evaluated in one step. After the evaluation of $\mathtt{s}$, the variable $\mathtt{v}$ has the same value as $\mathtt{e}$ before the evaluation, and all other variables remain unchanged. Hence:

$$[\![\mathtt{s}]\!] := Update(v, e, end_{\mathtt{s}}, start_{\mathtt{s}}) \tag{2.16}$$

Array assignments. Let $\mathtt{s}$ be an assignment $\mathtt{a[e_1] = e_2}$, where $\mathtt{a}$ is an array variable and $\mathtt{e_1}, \mathtt{e_2}$ are expressions. We consider that the assignment is evaluated in one step. After the evaluation of $\mathtt{s}$, the array $\mathtt{a}$ has the same value as before the evaluation, except for the position *pos* corresponding to the value of $\mathtt{e_1}$ before the evaluation, where the array now has the value of $\mathtt{e_2}$ before the evaluation. All other program variables remain unchanged and we have:

$$[\![\mathtt{s}]\!] := UpdateArr(v, e_1, e_2, end_{\mathtt{s}}, start_{\mathtt{s}}) \tag{2.17}$$

Conditional if-then-else Statements. Let $\mathtt{s}$ be the statement `if(Cond){c1} else {c2}`. The semantics of $\mathtt{s}$ states that entering the if-branch and/or entering the else-branch does not change the values of the variables:

$$[\![\mathtt{s}]\!] := [\![\mathtt{Cond}]\!](start_{\mathtt{s}}) \rightarrow EqAll(start_{\mathtt{c_1}}, start_{\mathtt{s}}) \tag{2.18a}$$

$$\wedge \quad \neg[\![\mathtt{Cond}]\!](start_{\mathtt{s}}) \rightarrow EqAll(start_{\mathtt{c_2}}, start_{\mathtt{s}}) \tag{2.18b}$$

While-Loops. Let s be the while-statement `while(Cond){c}`. We refer to Cond as the *loop condition*. We use the following three properties to define the semantics of s: (i) the iteration $lastIt_s$ is the first iteration where the loop condition does not hold, (ii) entering the loop body does not change the values of the variables, (iii) 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 $lastIt_s$. We then have:

$$[\![\mathtt{s}]\!] := \quad \forall it^{s\mathbb{D}}.\ (it^s < lastIt_{\mathtt{s}} \rightarrow [\![\mathtt{Cond}]\!](tp_{\mathtt{s}}(it^s))) \tag{2.19a}$$

$$\wedge \quad \neg[\![\mathtt{Cond}]\!](tp_{\mathtt{s}}(lastIt_{\mathtt{s}})) \tag{2.19b}$$

$$\wedge \quad \forall it^{s\mathbb{D}}.\ (it^s < lastIt_{\mathtt{s}} \rightarrow EqAll(start_{\mathtt{c}}, tp_{\mathtt{s}}(it^s)) \tag{2.19c}$$

$$\wedge \quad EqAll(end_{\mathtt{s}}, tp_{\mathtt{s}}(lastIt_{\mathtt{s}})) \tag{2.19d}$$

2.4.3 Partial Correctness using Axiomatic Semantics

Let $\mathtt{p}_0$ be a program, and let F be a property of $\mathtt{p}_0$ expressed in $\mathcal{L}$. We use the axiomatic semantics $[\![\mathtt{p}_0]\!]$ to establish that F is partially correct with respect to $\mathtt{p}_0$, by proving

$$[\![\mathtt{p}_0]\!] \vDash_{\mathbb{DUI}} F.$$

2.5 Soundness of Axiomatic Semantics

In this subsection, we show that the axiomatic semantics introduced in Section 2.4 is sound with respect to the operational semantics introduced in Section 2.3. Soundness is captured as follows:

2.11. Definition ($\mathcal{W}$-Soundness) Let $\mathtt{p}_0$ be a program and let F be a trace logic formula. Then F is called *$\mathcal{W}$-sound*, if for any execution-interpretation M of $\mathtt{p}_0$ we have $M \vDash F$.

The following theorem states that the axioms defining the predicate (*Reach*) are sound.

2.12. Theorem ($\mathcal{W}$-Soundness of axioms defining *Reach*) For a given terminating program p_0, the axioms defining *Reach* are $\mathcal{W}$-sound.

Proof. Let M be an execution interpretation.

First, let c be a context. Case distinction:

- Assume c is the top-level context. From $init^{sos}$ we conclude that $Reach(start_c)$ holds in M.
- Assume c occurs in an if-then-else-statement s := `if(Cond){c} else {c'}` and assume that both $\sigma Reach(start_{\mathtt{s}})$ and $\sigma[\![\mathtt{Cond}]\!](start_{\mathtt{s}})$ hold in M for some grounding σ of the enclosing iterations of s. Then rule ite_T^{sos} applies, from which we conclude that $\sigma Reach(start_{\mathtt{c}})$ holds in M.

- Assume c occurs in an if-then-else-statement s := `if(Cond){c'} else {c}`. Analogously to the previous case.
- Assume c is the context of the body of a while-statement s, and assume that $\sigma Reach(start_s)$ and $\sigma' it^s < \sigma lastIt_s$ hold in M for some grounding σ of the enclosing iterations of s and some grounding σ' of it^s. Using Lemma 2.9.3b and the fact $\sigma' it^s < \sigma lastIt_s$ we conclude that $\sigma\sigma' Reach(start_c)$ holds in M.

Second, let s be a non-while-statement occurring in context c. Assume further that $\sigma Reach(start_c)$ holds for some grounding σ of the enclosing iterations of c. Using Lemma 2.9.2, we conclude that $\sigma Reach(start_s)$ holds in M.

Third, let s be a while-statement occurring in context c. Assume further that both $\sigma Reach(start_c)$ and $\sigma' it^s \leq \sigma lastIt_s$ hold in M for some grounding σ of the enclosing iterations of c and some grounding σ of it^s. Using Lemma 2.9.2 we conclude that $\sigma\sigma' Reach(start_s)$ holds in M.

Finally, consider the last statement s of the top-level context c. From $init^{sos}$ we conclude that $Reach(start_c)$ holds in M. From this, we conclude $Reach(start_s)$ using Lemma 2.9.2. Finally, we apply Lemma 2.9.1 to conclude $Reach(end_s)$, which is the same as $Reach(end)$. □

We will now show that the axiomatic semantics of trace logic is $\mathcal{W}$-sound.

2.13. Theorem ($\mathcal{W}$-Soundness of Axiomatic Semantics of $\mathcal{W}$) For a given terminating program p_0, the semantics $[\![p_0]\!]$ is $\mathcal{W}$-sound.

Proof. Let M be an execution-interpretation of p_0. We have to show that for each statement s of p_0, the formula

$$\forall enclIts_s.(Reach(start_s) \rightarrow [\![s]\!])$$

holds in M. Let s now be an arbitrary statement of p_0. For an arbitrary grounding σ of the enclosing iterations assume that $\sigma Reach(start_s)$ holds in M. To show that $\sigma[\![s]\!]$ holds in M, we perform a case distinction on the type of the statement s:

- Let s be `skip`. Then $\sigma Reach(start_s)$ has been derived using $skip^{sos}$, so we know that $\sigma EqAll(start_s, end_s)$ holds in M, which is the same as $\sigma[\![s]\!]$.
- Let s be `v = e`. Then $\sigma Reach(start_s)$ has been derived using asg^{sos}, so we know that $\sigma Update(v, e, start_s, end_s)$ holds in M, which is the same as $\sigma[\![s]\!]$.
- Let s be `a[e1] = e2`. Then $\sigma Reach(start_s)$ has been derived using asg^{sos}_{arr}, so $\sigma UpdateArr(v, e_1, e_2, start_s, end_s)$ holds in M, which is the same as $\sigma[\![s]\!]$.
- Let s be `if(Cond){c1}else{c2}`. Assume that $\sigma[\![Cond]\!](start_s)$ holds in M. Using ite^{sos}_T, we conclude that $\sigma EqAll(start_{c_1}, tp_s)$ holds in M. In particular, (2.18a) holds.
  Analogously, we prove that (2.18b) holds in M. Combining both results, we conclude that $\sigma[\![s]\!]$ holds in M.

- Let s be `while(Cond){p1}`. Formulas (2.19a) and (2.19b) define $\sigma lastIt_s$ as the smallest iteration it where $\sigma[\![\texttt{Cond}]\!](tp_{\mathrm{s}}(it)$ does not hold in M. Since we assume termination, such an iteration needs to exist, and in particular the definition is well-defined, so (2.19a) and (2.19b) hold in M.
  Now let it be an arbitrary iteration such that $it < \sigma lastIt_s$ holds in M. Using Lemma 2.9.3a, we conclude that $\sigma Reach(tp_{\mathrm{s}}(it))$ holds in M from the fact that $\sigma Reach(start_{\mathrm{s}})$ holds in M. Since $\sigma[\![\texttt{Cond}]\!](tp_{\mathrm{s}}(it))$ holds in M by the assumption $it < \sigma lastIt_s$, we know that $\sigma Reach(start_{\mathrm{s}})$ has been derived using $while_T^{sos}$, and in particular that $\sigma EqAll(start_c, tp_{\mathrm{s}}(it)$ holds in M, which is the same as (2.19c).
  Finally, we obtain that $\sigma Reach(tp_{\mathrm{s}}(lastIt_s))$ holds in M from the fact that $\sigma Reach(start_{\mathrm{s}})$ holds in M using Lemma 2.9.3a. By definition of $lastIt_s$, the formula $\sigma[\![\texttt{Cond}]\!](tp_{\mathrm{s}}(lastIt_s))$ does not hold in M, so $\sigma Reach(start_{\mathrm{s}})$ has been derived using $while_F^{sos}$. In particular, $\sigma EqAll(end_{\mathrm{s}}, tp_{\mathrm{s}}(lastIt_{\mathrm{s}}))$ holds in M, which is the same as (2.19d).

□

2.6 Completeness of Axiomatic Semantics

Developed more than 50 years ago, Hoare logic [Hoa69] is the standard framework to reason axiomatically about programs and forms the basis of several verification frameworks, including ESC/Java [FLL+02], Spec# [BLS04], HAVOC [CLQR07], and VCC [CDH+09].

In this section, we clarify the relation between Hoare logic and the axiomatic semantics of trace logic. We present a meaningful and simple translation, which maps for any given program $\mathtt{p}_0$ Hoare triples to trace logic formulas, with the property that for any valid Hoare triple, the resulting trace logic formula follows from $[\![\mathtt{p}_0]\!]$ in $\mathbb{D} \cup \mathbb{I}$. As a consequence, we obtain that trace logic is complete with respect to Hoare logic (for a meaningful definition of completeness). In other words, whenever we can conclude the validity of a given program property using Hoare logic, we can also conclude the validity of that property using trace logic.

2.6.1 Hoare Logic

In this subsection, we recall the standard definition of Hoare logic [Hoa69].

2.14. Definition (Hoare logic) Let $\mathtt{p}_0$ be a fixed program.

- The *single-state-language* of $\mathtt{p}_0$ is an instance of first-order logic, with a signature consisting of $S_{\mathbb{I}}$ and a nullary function symbol x for each program variable $\mathtt{x}$ of $\mathtt{p}_0$.
- The *language of Hoare logic* of $\mathtt{p}_0$ consists of so-called *Hoare triples* of the form $\{F\}p\{G\}$, for any subprogram $\mathtt{p}$ of $\mathtt{p}_0$ and single-state-language formulas F and G.
- A Hoare triple is called *valid*, if it can be derived from the rules of the Hoare logic calculus, denoted in Figure 2.5.

$$\text{ass}\ \frac{}{\{F[x \mapsto e]\}x := e\{F\}}$$

$$\text{skip}\ \frac{}{\{F\}skip\{F\}}$$

$$\text{Weakening}\ \frac{F_1 \to F_1' \qquad \{F_1'\}p\{F_2'\} \qquad F_2' \to F_2}{\{F_1\}p\{F_2\}}$$

$$\text{Concatenation}\ \frac{\{G_1\}p_1\{G_2\} \qquad \dots \qquad \{G_k\}p_k\{G_{k+1}\}}{\{G_1\}p_1;\dots;p_k\{G_{k+1}\}}$$

$$\text{ite}\ \frac{\{[\![\texttt{Cond}]\!] \wedge F_1\}p_1\{F_2\} \qquad \{\neg[\![\texttt{Cond}]\!] \wedge F_1\}p_2\{F_2\}}{\{F_1\}\ \texttt{if(Cond)\{p}_1\texttt{\}else\{p}_2\texttt{\}}\{F_2\}}$$

$$\text{while}\ \frac{\{[\![\texttt{Cond}]\!] \wedge F\}p_1\{F\}}{\{F\}\texttt{while(Cond)\{p}_1\texttt{\}}\{\neg[\![\texttt{Cond}]\!] \wedge F\}}$$

Figure 2.5: Proof system of Hoare logic.

If we compare how program variables are formalized in the single-state-language of Hoare logic and in trace logic, we see that timepoints are missing in the single-state-language. To compensate for keeping timepoints only implicit and still being able to refer to values of the program variables at (two) different timepoints, Hoare logic uses Hoare triples $\{F\}\mathtt{p}\{G\}$, with the intuitive meaning that if F holds at the timepoint where we start to execute p, then G holds at the timepoint after the execution of p. Note that the language of Hoare triples itself is not an instance of first-order logic. In particular, the rules weakening and concatenation are explicitly needed to simulate the transitivity of entailment in first-order logic.

2.6.2 Translating Hoare Triples to Trace Logic Formulas

We will now present a translation from Hoare triples to trace logic formulas. Recall that a Hoare triple $\{F_1\}\mathtt{p}\{F_2\}$ denotes that if F_1 holds at the start of p, then F_2 holds at the end of F_2. We write such a fact in trace logic as $[F_1](start_{\mathtt{p}}) \to [F_2](end_{\mathtt{p}})$, where the expressions $[F_1](start_{\mathtt{p}})$ and $[F_2](end_{\mathtt{p}})$ denote the result of adding to each program variable in F_1 resp. F_2 the timepoint $start_{\mathtt{p}}$ resp. $end_{\mathtt{p}}$ as first argument. For example, consider the program $\mathtt{p_0}$:= i=i+1. We can derive the Hoare triple $\{i \simeq 2\}\mathtt{p_0}\{i \simeq 3\}$. Similarily, we can derive the trace logic formula

$$i(start_{\mathtt{p_0}}) \simeq 2 \to i(end_{\mathtt{p_0}}) \simeq 3.$$

Additionally, we have to deal with the technical complication that Hoare logic overspecifies unreachable subprograms: Consider a program $\mathtt{p_0}$, containing p := i=i+1 as an

unreachable subprogram. As Hoare logic does not take the context of a subprogram into account, we can again derive the Hoare triple $\{i \simeq 2\}\mathtt{p}\{i \simeq 3\}$, even though $\mathtt{p}$ is never executed. In contrast, in trace logic we will only derive the more precise

$$\forall enclIts_p.\Big(Reach(start_{\mathtt{p}}) \rightarrow \big(i(start_{\mathtt{p}}) \simeq 2 \rightarrow i(end_{\mathtt{p}}) \simeq 3\big)\Big),$$

which takes the reachability of the subprogram $\mathtt{p}$ into account. Note that this difference only occurs for (strict) subprograms, as the start of a program is by definition always reachable.

2.15. Definition (Completeness with respect to Hoare logic) Let $\mathtt{p_0}$ be a fixed program.

- Let [] be a function which translates any Hoare logic formula F to the trace logic formula F', where F' is obtained by adding to each symbol v denoting a program variable in F as first argument the free variable $tp^{\mathbb{L}}$. For any background theory $\mathcal{T}$, let further $[\mathcal{T}] := \{\forall tp^{\mathbb{L}}.[F] \mid F \in \mathcal{T}\}$ be the translation of $\mathcal{T}$.
- Trace logic is called *complete with respect to Hoare logic*, if for any fixed background theory $\mathcal{T}$, for any subprogram $\mathtt{p}$ of $\mathtt{p_0}$ and for any Hoare triple $\{F_1\}\mathtt{p}\{F_2\}$ provable in $\mathcal{T}$, the trace logic formula

  $$\forall enclIts_p.\Big(Reach(start_{\mathtt{p}}) \rightarrow \big([F_1](start_{\mathtt{p}}) \rightarrow [F_2](end_{\mathtt{p}})\big)\Big)$$

  follows from the axiomatic semantics $[\![\mathtt{p_0}]\!]$ of trace logic in $[\mathcal{T}]$.

2.16. Theorem Trace logic is complete with respect to Hoare logic.

Proof. Let $\mathtt{p_0}$ be a fixed terminating program. We proceed by structural induction on the Hoare calculus derivation with the induction hypothesis that for any subprogram $\mathtt{p}$ of $\mathtt{p_0}$ and for any formulas F_1, F_2, if $\{F_1\}p\{F_2\}$ is derivable in the Hoare Calculus, then

$$\forall enclIts_p.\Big(Reach(start_{\mathtt{p}}) \rightarrow \big([F_1](start_{\mathtt{p}}) \rightarrow [F_2](end_{\mathtt{p}})\big)\Big)$$

is entailed by the trace logic semantics.

Consider now an arbitrary subprogram $\mathtt{p}$ of $\mathtt{p_0}$ such that $\{F_1\}p\{F_2\}$ is derivable in Hoare logic. For an arbitrary grounding σ of the enclosing iterations assume that $\sigma Reach(start_{\mathtt{p}})$ holds. This fact together with the definition of the trace logic semantics implies that $\sigma[\![\mathtt{p}]\!]$ holds. We now use a case distinction to show the implication

$$\sigma[F_1](start_{\mathtt{p}}) \rightarrow \sigma[F_2](end_{\mathtt{p}}). \tag{2.20}$$

Since the grounding σ is arbitrary, this then concludes the proof.

- Skip: Assume the last rule is

  $$\frac{}{\{F_1\}skip\{F_1\}}$$

We have to show $\sigma[F_1](start_{\mathrm{p}}) \to \sigma[F_1](end_{\mathrm{p}})$. The semantics $\sigma[\![\mathrm{p}]\!]$ state that $\sigma EqAll(start_{\mathrm{p}}, end_{\mathrm{p}})$ holds. Using this formula, we can rewrite $\sigma[F_1](start_{\mathrm{p}})$ into $\sigma[F_1](end_{\mathrm{p}})$, which shows that (2.20) holds.

- Assignment: Assume that the last rule is

$$\frac{}{\{F_2[x \mapsto e]\}x := e\{F_2\}}$$

We have to show the implication $\sigma[F_2][x \mapsto e](start_{\mathrm{p}}) \to \sigma[F_2](end_{\mathrm{p}})$. By definition, $\sigma[\![\mathrm{p}]\!]$ consists of $\sigma(x(end_{\mathrm{p}})) = \sigma([\![e]\!](start_{\mathrm{p}}))$ and of $\sigma(v(end_{\mathrm{p}})) = \sigma(v(start_{\mathrm{p}}))$ for all other variables v. Using these equations, we rewrite $\sigma[F_2][x \mapsto e](start_{\mathrm{p}})$ into $\sigma[F_2](end_{\mathrm{p}})$, which shows that (2.20) holds.

- Weakening: Assume the last rule is

$$\frac{F_1 \to F_1' \qquad \{F_1'\}p\{F_2'\} \qquad F_2' \to F_2}{\{F_1\}p\{F_2\}}$$

First, the formulas $F_1 \to F_1'$ and $F_2' \to F_2$ are tautologies in Hoare logic. Since we assume that [] maps Hoare logic tautologies to trace logic tautologies, we get that $[F_1 \to F_1'](tp)$ and $[F_2' \to F_2](tp)$ hold for arbitrary ground timepoints tp. In particular, $[F_1 \to F_1'](\sigma(start_{\mathrm{p}}))$ and $[F_2' \to F_2](\sigma(end_{\mathrm{p}}))$ hold, which can be written as $\sigma[F_1](start_{\mathrm{p}}) \to \sigma[F_1'](start_{\mathrm{p}})$ and $\sigma[F_2'](end_{\mathrm{p}}) \to \sigma[F_2](end_{\mathrm{p}})$. Second, we use the induction hypothesis and the assumption $\sigma Reach(start_{\mathrm{p}})$ to conclude that the trace logic axioms imply $\sigma[F_1'](start_{\mathrm{p}}) \to \sigma[F_2'](end_{\mathrm{p}})$. Combining the three implications shows that (2.20) holds.

- Concatenation: Assume the last rule is

$$\frac{\{G_1\}p_1\{G_1'\} \qquad \dots \qquad \{G_k\}p_k\{G_k'\}}{\{G_1\}p_1;\dots;p_k\{G_k'\}}$$

where $G_1 = F_1$ and $G_k' = F_2$. Using Lemma 2.9.2, we conclude from $\sigma Reach(start_{\mathrm{p}})$ that $\sigma Reach(start_{\mathrm{p_i}})$ holds for any $1 \leq i \leq k$. Combining these facts with applications of the induction hypothesis yields that $\sigma[G_i](start_{\mathrm{p_i}}) \to \sigma[G_i'](end_{\mathrm{p_i}})$ holds for any $1 \leq i \leq k$. Since $\sigma(end_{\mathrm{p_i}}) = \sigma(start_{\mathrm{p_{i+1}}})$ for any $1 \leq i < k$, we use a trivial subinduction to conclude

$$\sigma[G_1](start_{\mathrm{p}}) \to \sigma[G_k'](end_{\mathrm{p}}).$$

In particular, since $G_1 = F_1$ and $G_k' = F_2$, we conclude that (2.20) holds.

- ITE: Assume that the last rule is

$$\frac{\{[\![\mathtt{Cond}]\!] \wedge F_1\}p_1\{F_2\} \qquad \{\neg[\![\mathtt{Cond}]\!] \wedge F_1\}p_2\{F_2\}}{\{F_1\}\ \mathtt{if(Cond)\{p_1\}else\{p_2\}}\{F_2\}}$$

W.l.o.g. assume that $\sigma[\![\mathtt{Cond}]\!](start_{\mathrm{p}})$ holds. We assume that $\sigma[F_1](start_{\mathrm{p}})$ holds with the goal of deriving $\sigma[F_2](end_{\mathrm{p}})$. First, we combine $\sigma[\![\mathrm{p}]\!]$ with $\sigma[\![\mathtt{Cond}]\!](start_{\mathrm{p}})$ to derive $\sigma EqAll(start_{\mathrm{p}}, start_{\mathrm{p_1}})$. From this we derive $\sigma[\![\mathtt{Cond}]\!](start_{\mathrm{p_1}})$ and $\sigma[F_1](start_{\mathrm{p_1}})$.

Second $\sigma Reach(start_{\mathrm{p}})$ and $\sigma[\![\texttt{Cond}]\!](start_{\mathrm{p}})$ imply $\sigma Reach(start_{\mathrm{p}_1})$. We then combine the induction hypothesis

$$\sigma Reach(start_{\mathrm{p}_1}) \rightarrow \Big(\sigma([\![\texttt{Cond}]\!] \wedge [F_1])(start_{\mathrm{p}_1}) \rightarrow \sigma[F_2](end_{\mathrm{p}_1})\Big)$$

with $\sigma Reach(start_{\mathrm{p}_1})$, $\sigma[\![\texttt{Cond}]\!](start_{\mathrm{p}_1})$ and $\sigma[F_1](start_{\mathrm{p}_1})$ to obtain $\sigma[F_2](end_{\mathrm{p}_1})$. Since $end_{\mathrm{p}_1} = end_{\mathrm{p}}$, we conclude $\sigma[F_2](end_{\mathrm{p}})$, which shows that (2.20) holds.

- While: Assume that the last rule is

$$\frac{\{[\![\texttt{Cond}]\!] \wedge F\}p_1\{F\}}{\{F\}\texttt{while(Cond)\{p}_1\texttt{\}}\{\neg[\![\texttt{Cond}]\!] \wedge F\}}$$

We again assume that $\sigma[F_1](start_{\mathrm{p}})$ holds with the goal of deriving $\sigma[F_1](end_{\mathrm{p}})$. We perform a bounded sub-induction on it^p from 0 to $\sigma lastIt_{\mathrm{p}}$ with the induction hypothesis $\sigma[F_1](tp_{\mathrm{p}}(it^p))$.
Base Case: The formula $\sigma[F_1](start_{\mathrm{p}})$ holds and can be written as $\sigma[F_1](tp_{\mathrm{p}}(\texttt{0}))$.
Inductive Case: We have to show the implication

$$\sigma\forall it^p.\Big((it^p < lastIt_{\mathrm{p}} \wedge [F_1](tp_{\mathrm{p}}(it^p))) \rightarrow [F_1](tp_{\mathrm{p}}(\texttt{succ}(it^p)))\Big).$$

Let σ' be an extension of σ with an arbitrary grounding of it^p, and assume that $\sigma'(it^p < lastIt_{\mathrm{p}})$ and $\sigma'[F_1](tp_{\mathrm{p}}(it^p))$ hold. We now have to show $\sigma'[F_1](tp_{\mathrm{p}}(\texttt{succ}(it^p)))$.
Combining $\sigma'[\![\mathrm{p}]\!]$ and $\sigma'(it^p < lastIt_{\mathrm{p}})$ yields both $\sigma'[\![\texttt{Cond}]\!](tp_{\mathrm{p}}(it^p))$ and $\sigma' EqAll(start_{\mathrm{p}_1}, tp_{\mathrm{p}}(it^p))$. We use the latter fact first to rewrite the former fact to $\sigma'[\![\texttt{Cond}]\!](start_{\mathrm{p}_1})$ and second to rewrite $\sigma'[F_1](tp_{\mathrm{p}}(it^p))$ to $\sigma'[F_1](start_{\mathrm{p}_1})$. Third, we obtain $\sigma' Reach(start_{\mathrm{p}_1})$ using Lemma 2.9.3b.
The induction-hypothesis now states

$$\forall enclIts_p.\Big(Reach(start_{\mathrm{p}_1}) \rightarrow \Big(([\![\texttt{Cond}]\!] \wedge [F_1])(start_{\mathrm{p}_1}) \rightarrow [F_1](end_{\mathrm{p}_1})\Big)\Big).$$

For the grounding σ' we have already established the three premises of this formula, therefore we conclude $\sigma'[F_1](end_{\mathrm{p}_1})$. Since $end_{\mathrm{p}_1} = tp_{\mathrm{p}}(\texttt{succ}(it^p))$, we get $\sigma'[F_1](tp_{\mathrm{p}}(\texttt{succ}(it^p)))$, which concludes the inductive case.
We now have established the base case and the inductive case, so we use bounded induction to conclude $\sigma[F_1](tp_{\mathrm{p}}(lastIt_{\mathrm{p}}))$. Finally, we rewrite this fact to $\sigma[F_1](end_p)$ using $\sigma'[\![\mathrm{p}]\!]$, which shows that (2.20) holds.

□

2.7 Related Work

Most frameworks, which enable formal proofs about software program properties, are based on a reasoning language that encodes *single* states/configurations and properties about them. This includes Hoare's seminal work on axiomatic semantics [Hoa69], and

the state-of-the-art approach from [RES10]. To prove properties about (potentially unbounded) executions, these approaches introduce meta-level proof rules [Hoa69, RS12], in a way such that each property, which is derivable for a given program, is (partially) correct for that program. In contrast to these approaches, trace logic formalizes the semantics of (potentially unbounded) executions directly as first-order axioms using standard first-order logic semantics, without introducing any meta-level proof rules. This is possible since trace logic captures each program variable as a function whose value depends on the current timepoint in the execution, in combination with universal quantification over iterations. As a result, (i) each model of our axiomatization corresponds to a single (potentially unbounded) execution of the program, and (ii) we can leverage general-purpose techniques of first-order logic, including off-the-shelf automated theorem provers and proof theory.

In [BS01, BB13], the semantics of programs are encoded into an instance of first-order modal logic. Afterwards, the correctness of properties is established in a custom sequent calculus. Compared to [BS01, BB13], modal operators are not needed in trace logic, and properties in trace logic can be proven in any sound proof system for first-order logic.

CHAPTER 3

Software Verification using Trace Logic

In this chapter, we present a new verification framework for software programs including loops and arrays based on trace logic. We will first motivate the design of our framework by highlighting the challenges of verifying properties of programs containing loops and arrays on the running example. In particular, we will argue that advanced loop splitting is necessary to verify the given property, and we will highlight how the expressiveness of trace logic in combination with explicit induction axioms enables such advanced automated loop splitting. Afterwards, we will present our verification framework in detail, including a formal presentation of the inductive consequences we use in our work, the so-called *trace lemmas*. Finally, we show how to formally prove the property of the running example in our verification framework.

3.1 Key Ideas

3.1.1 Motivating Example

Recall the property (2.1), which we want to prove correct with respect to our running example from Figure 2.1. This property is challenging to verify, since it requires theory-specific reasoning over integers and it involves an alternation of quantifiers, as the length of the array `b` is unbounded and the position *apos* is arbitrary.

To understand the difficulty in verifying such kind of properties, let us first illustrate how humans would naturally prove property (2.1). First, prove that for each position *bpos* of the array `b` there *exists a loop iteration* it_0 at which *bpos* is visited. In particular, the value of `b` at *bpos* at the beginning of the successor-iteration of it_0 is v, where v is the value of `a` at the position corresponding to the value of `i` in the loop iteration it_0. Then, *split the iterations of the loop* into two intervals: (i) The interval from the

first iteration of the loop to the successor-iteration of it_0, and (ii) the interval from the successor-iteration of it_0 to the last iteration of the loop. Finally, prove that the value v of `b` at position *bpos* is preserved *throughout the second interval*: In particular, one uses *(bounded) inductive reasoning*, to conclude the *preservation* of v across the whole second interval from the step-wise preservation of v in that interval.

While the above proof might be natural for humans, it is challenging for automated verifiers for the following reasons:

(C1) one needs to express and relate different iterations in the execution of the loop in Figure 2.1,
(C2) one needs to automatically synthesize loop iteration terms, which depend on values of program variables and therefore potentially denote different iterations for different program executions, and
(C3) one needs to split the loop into intervals using the synthesized loop iteration terms and reason about the resulting loop intervals separately.

In the remainder of this chapter, we will introduce a new verification framework based on trace logic, which addresses these challenges.

Trace logic is able to express and relate iterations of loops, and therefore able to address the challenge (C1), as follows. First, in trace logic, program variables are encoded as unary and binary functions over program execution timepoints. This way, we can precisely express the value of each program variable at any program execution timepoint, without introducing abstractions. For Figure 2.1, for example, we write $i(l_9(\mathtt{0}))$ to denote the value of `i` at timepoint $l_9(\mathtt{0})$. Secondly, trace logic allows arbitrary quantification over iterations and values of program variables. In particular, we can express and reason about iterations that depend on (possibly non-ground) expressions involving program variables.

Moreover, trace logic can express induction axioms, including bounded and generalized bounded induction axioms as described in Section 2.1. It therefore enables reasoning with these induction axioms directly in the language. In the following two subsections, we will show that this is the key to enable automated loop splitting, that is, to address the challenges (C2) and (C3).

3.1.2 Synthesizing Loop Iteration Terms

We will now show how to use (generalized) bounded induction axioms to synthesize complex loop iteration terms, which can be used as interval-boundaries for splitting loops. Consider an induction hypothesis IH with free variable it, and two terms it_L and it_R of sort $\mathbb{D}$. Recall from Section 2.1 the bounded induction axiom for IH, which is

$$\Big(BC(it_L) \wedge IC(it_L, it_R)\Big) \rightarrow Concl(it_L, it_R). \tag{3.1}$$

Assume now that $BC(it_L)$ and $\neg Concl(it_L, it_R)$ hold. We can then combine these two facts with (3.1) to derive

$$\neg IC(it_L, it_R),$$

as follows: Since the conclusion $Concl(it_L, it_R)$ of (3.1) does not hold, we use backward-style reasoning to conclude that one of the two premises of (3.1) must not hold. As we also know that the first premise $BC(it_L)$ holds, we conclude that the second premise $IC(it_L, it_R)$ must not hold. Note that we were able to use the induction axiom (3.1) even though the inductive case does not hold, by *reasoning backward*. It would not have been possible to apply the induction axiom (3.1) using forward-style reasoning, as the inductive case does not hold.

We see the actual power of deducing the consequence $\neg Concl(it_L, it_R)$ as soon as we inline its definition, which gives us

$$\exists it_1^{\mathbb{D}}.\Big(IH(it_1) \wedge \neg IH(\mathtt{succ}(it_1))\Big).$$

By applying backward reasoning, we were able to automatically synthesize a new existentially quantified iteration term it_1, for which we know that the inductive step does not hold. Note that it_1 can denote different iterations in different executions of the program. In particular, if we use it_1 to split the loop, then the loop is potentially split at different iterations for different program executions.

3.1. Example Let `w` be a top-level while-statement, and let `v` be an integer program variable which is incremented at most by `1` in each iteration of `w`. Recall the definitions of $tp_{\mathtt{w}}$ and $lastIt_{\mathtt{w}}$ from Subsection 2.2.1. Consider now the induction-hypothesis $v(tp_{\mathtt{w}}(it)) \leq 1$ with free variable it, the two terms `0` and $lastIt_{\mathtt{w}}$, and the corresponding bounded induction axiom

$$\Big(BC(\mathtt{0}) \wedge IC(\mathtt{0}, lastIt_{\mathtt{w}})\Big) \rightarrow Concl(\mathtt{0}, lastIt_{\mathtt{w}}),$$

where BC, IC, and $Concl$ are defined as follows:

$$BC(\mathtt{0}) := v(tp_{\mathtt{w}}(\mathtt{0})) \leq 1 \tag{3.2}$$

$$IC(\mathtt{0}, lastIt_{\mathtt{w}}) := \forall it^{\mathbb{D}}.\Big((\mathtt{0} \leq it < lastIt_{\mathtt{w}} \wedge v(tp_{\mathtt{w}}(it)) \leq 1) \rightarrow v(tp_{\mathtt{w}}(\mathtt{succ}(it))) \leq 1\Big) \tag{3.3}$$

$$Concl(\mathtt{0}, lastIt_{\mathtt{w}}) := \forall it^{\mathbb{D}}.\Big(\mathtt{0} \leq it \leq lastIt_{\mathtt{w}} \rightarrow v(tp_{\mathtt{w}}(it)) \leq 1\Big) \tag{3.4}$$

Assume now that $v(tp_{\mathtt{w}}(\mathtt{0})) \simeq 0$ and $v(tp_{\mathtt{w}}(lastIt_{\mathtt{w}})) \not\leq 1$ hold. From these two facts we immediately conclude that $BC(\mathtt{0})$ and $\neg Concl(\mathtt{0}, lastIt_{\mathtt{w}})$ hold. Combining those facts with the induction axiom gives $\neg IC(\mathtt{0}, lastIt_{\mathtt{w}})$, which, by definition, is

$$\exists it_1^{\mathbb{D}}.\Big(\mathtt{0} \leq it_1 < lastIt_{\mathtt{w}} \wedge v(tp_{\mathtt{w}}(it_1)) \leq 1 \wedge v(tp_{\mathtt{w}}(\mathtt{succ}(it_1))) \not\leq 1\Big).$$

We then use further integer theory reasoning to conclude $\exists it_1^{\mathbb{D}}.v(tp_{\mathtt{w}}(it_1)) \simeq 1$. Taking a step back, we see that we were able to combine an induction axiom with basic facts to synthesize the iteration it_1, for which we know that `v` has value 1. □

3.1.3 General Lemmas and Instantiating Them to Parts of Loops

Next, we will show how generalized bounded induction axioms can be used to reason about loop intervals separately, by instantiating universally quantified variables over iterations with concrete interval bounds. Consider an induction hypothesis IH with free variables $it, x_1, \ldots, x_k$, and recall from Section 2.1 the generalized bounded induction axiom for IH, which is

$$\forall x_1^{\mathbb{I}}, \ldots, x_k^{\mathbb{I}}, it_L^{\mathbb{D}}, it_R^{\mathbb{D}}. \Big(\big(BC(it_L) \wedge IC(it_L, it_R) \big) \rightarrow Concl(it_L, it_R) \Big)$$

Assume now that it_1 and it_2 denote two iterations such that $BC(tp_1)$ and $IC(tp_1, tp_2)$ hold. We can then instantiate it_L and it_R in the generalized bounded induction axiom to tp_1 resp. tp_2, and resolve it with $BC(tp_1)$ and $IC(tp_1, tp_2)$ to conclude $Concl(it_1, it_2)$. While the generalized induction axiom is formulated for arbitrary intervals, we apply it only to the interval with bounds tp_1 and tp_2. This allows us to use the induction axiom, even though the inductive case might not hold for the whole loop. In our experience, it is often the case that the induction case holds for each timepoint in a given interval of a loop, but does not hold for each timepoint of the whole loop.

Note that we do not need to add the right instantiations of these induction axioms upfront to the search space. Instead, we add universally quantified versions of these induction axioms and instantiate them on demand during proof search. This is important, as the terms, which we instantiate these axioms with, are usually only synthesized during proof search, and therefore not available upfront.

3.2. Example Let `w` be a top-level while-statement, let `a` be a constant array variable, and let `b` be a (mutable) array variable. Consider now the induction-hypothesis $b(tp_{\mathtt{w}}(it), bpos) \simeq v$ with free variables $it^{\mathbb{D}}$, $bpos^{\mathbb{I}}$ and $v^{\mathbb{I}}$, and the corresponding generalized bounded induction axiom

$$\forall bpos^{\mathbb{I}}, v^{\mathbb{I}}, it_L^{\mathbb{D}}, it_R^{\mathbb{D}}. \Big(\big(BC(it_L) \wedge IC(it_L, it_R) \big) \rightarrow Concl(it_L, it_R) \Big)$$

where BC, IC, and $Concl$ are defined as follows:

$$BC(it_L) := b(tp_{\mathtt{w}}(it_L), bpos) \simeq v \tag{3.5}$$

$$IC(it_L, it_R) := \forall it^{\mathbb{D}}. \Big((it_L \leq it < it_R \wedge b(tp_{\mathtt{w}}(it), bpos) \simeq v) \rightarrow b(tp_{\mathtt{w}}(\mathtt{succ}(it)), bpos) \simeq v \Big) \tag{3.6}$$

$$Concl(it_L, it_R) := \forall it^{\mathbb{D}}. \Big(it_L \leq it \leq it_R \rightarrow b(tp_{\mathtt{w}}(it), bpos) \simeq v \Big) \tag{3.7}$$

Assume now that we already know for some (previously synthesized) timepoint it_1 and positions pos_1,pos_2 that the following two facts hold:

$$b(tp_{\mathtt{w}}(it_1), pos_1) \simeq a(pos_2)$$

$$\forall it^{\mathbb{D}}.\Big((it_1 \leq it < lastIt_{\mathtt{w}} \wedge b(tp_{\mathtt{w}}(it), pos_1) \simeq a(pos_2)) \rightarrow b(tp_{\mathtt{w}}(\mathtt{succ}(it)), pos_1) \simeq a(pos_2)\Big).$$

We can then instantiate it_L, it_R, $bpos$ and v in the induction axiom to it_1, $lastIt_{\mathtt{w}}$, pos_1, and $a(pos_2)$, to derive

$$\forall it^{\mathbb{D}}.\Big(it_1 \leq it \leq lastIt_{\mathtt{w}} \rightarrow b(tp_{\mathtt{w}}(it), pos_1) \simeq a(pos_2)\Big),$$

from which we finally conclude $b(tp_{\mathtt{w}}(lastIt_{\mathtt{w}}), pos_1) \simeq a(pos_2)$. □

3.2 A Verification Framework Based on Trace Logic

Recall from Section 2.4 that trace logic formulates the axiomatic semantics of a program as a set of axioms in standard first-order logic modulo the combined background theory $\mathbb{D} \cup \mathbb{I}$ of difference logic and integers. This way, establishing the correctness of a property of a program is reduced to a validity check in standard first-order logic modulo $\mathbb{D} \cup \mathbb{I}$. We can therefore use any off-the-shelf first-order theorem prover, which supports validity-queries over (fully-quantified) standard first-order logic modulo $\mathbb{D} \cup \mathbb{I}$, to reason about the correctness of program properties. In particular, we can use saturation-based theorem provers, including those implementing the superposition-calculus [KV13, Sch02, WDF+09], and SMT-solvers, for instance, solvers based on some refinement of CDCL [DMB08, BCD+11].

Unfortunately, we cannot expect a theorem prover in practice to establish the correctness of a property if we only pass it the axiomatic semantics of the program and the property without any further information: As we have motivated in Section 3.1, inductive reasoning over loop iterations and automated loop splitting are required to derive many relevant consequences of the semantics, since we target programs containing unbounded loops. Automating such kind of induction is however challenging: state-of-the-art theorem-provers, including both saturation-based provers and SMT-solvers, are not able to automatically infer the (generalized) bounded induction axioms needed in the setting of trace logic. To address this problem, we (i) identified some of the most important applications of induction that are useful for many programs, (ii) formulated the corresponding inductive properties in trace logic as *trace lemmas*, and (iii) use these trace lemmas to guide the reasoning of the theorem prover while reasoning about the correctness of a given property.

Summing up, our verification framework works as follows. Assume we are given a $\mathcal{W}$-program $\mathtt{p}_0$ and a property F formulated in trace logic. The workflow to establish the (partial) correctness of F is visualized in Figure 3.1. First, we generate the axiomatic semantics $[\![\mathtt{p}_0]\!]$ of $\mathtt{p}_0$ in trace logic, as described in Section 2.4. Secondly, we generate a set of lemmas L formulated in trace logic – the so-called *trace lemmas* – in order to cover the inductive reasoning necessary to prove the correctness of F. Thirdly, we pass the semantics $[\![\mathtt{p}_0]\!]$, the trace lemmas L and the property F to a theorem prover and ask the prover to prove the validity of $[\![\mathtt{p}_0]\!] \wedge L \models_{\mathbb{D}\cup\mathbb{I}} F$. If the prover finds a proof, we conclude that F is partially correct with respect to $\mathtt{p}_0$.

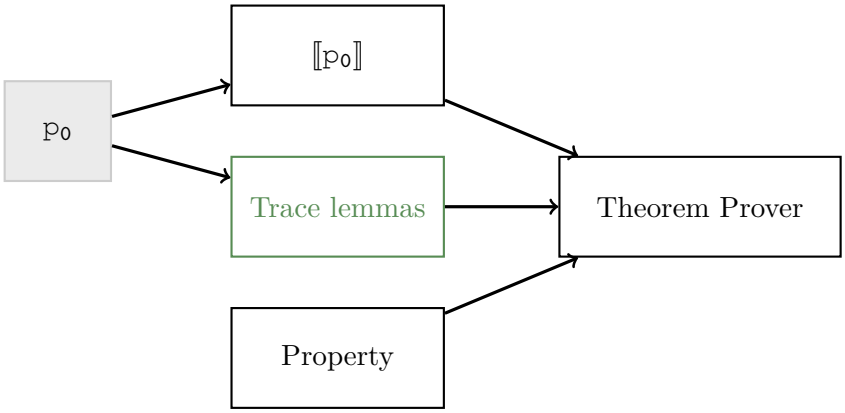

Figure 3.1: Workflow of the verification framework.

3.2.1 Trace Lemmas for Reasoning About Arrays

In this subsection, we describe the set of trace lemmas we use to reason about the partial correctness of properties of programs including loops and arrays. Each trace lemma is a consequence of the (generalized) bounded induction axioms of difference logic and other axioms of the combined theory $\mathbb{D} \cup \mathbb{I}$, except for the At-Least-One-Iteration trace lemma, which is a consequence of the axiomatic semantics. We generate the trace lemmas for each applicable variable and each loop of the program using a simple static analysis of the given program. In the remainder of this subsection, we describe each lemma, prove it formally, and motivate its usefulness.

Value-Evolution trace lemma. Let $\mathtt{w}$ be a while-statement, $\mathtt{v}$ be a mutable program variable, and $\trianglelefteq$ be a reflexive and transitive predicate (e.g. $\simeq$, $\leq$ or $\geq$). If $\mathtt{v}$ is an integer program variable, then the *Value-Evolution trace lemma for* $\mathtt{w}$, $\mathtt{v}$ *and* $\trianglelefteq$ is

$$\begin{aligned}
\forall enclIts. \forall it_L^{\mathbb{D}}, it_R^{\mathbb{D}}. \Big(& \\
& \forall it^{\mathbb{D}}. \Big((it_L \leq it < it_R \wedge v(tp_{\mathtt{w}}(it_L)) \trianglelefteq v(tp_{\mathtt{w}}(it))) \rightarrow \\
& \quad v(tp_{\mathtt{w}}(it_L)) \trianglelefteq v(tp_{\mathtt{w}}(\mathtt{succ}(it))) \Big) \\
& \rightarrow \\
& \Big(it_L \leq it_R \rightarrow v(tp_{\mathtt{w}}(it_L)) \trianglelefteq v(tp_{\mathtt{w}}(it_R)) \Big) \Big)
\end{aligned}$$

If $\mathtt{v}$ is an array variable, then the *Value-Evolution trace lemma for* $\mathtt{w}$, $\mathtt{v}$ *and* $\trianglelefteq$ is

$$\forall enclIts. \forall it_L^{\mathbb{D}}, it_R^{\mathbb{D}}, pos^{\mathbb{I}}. \Big(\\ \forall it^{\mathbb{D}}. \Big((it_L \leq it < it_R \wedge v(tp_{\mathtt{w}}(it_L), pos) \trianglelefteq v(tp_{\mathtt{w}}(it), pos)) \rightarrow \\ v(tp_{\mathtt{w}}(it_L), pos) \trianglelefteq v(tp_{\mathtt{w}}(\mathtt{succ}(it)), pos) \Big) \\ \rightarrow \\ \Big(it_L \leq it_R \rightarrow v(tp_{\mathtt{w}}(it_L), pos) \trianglelefteq v(tp_{\mathtt{w}}(it_R), pos) \Big) \Big)$$

The Value-Evolution lemma instantiated with the equality predicate $\simeq$ is very useful, as it allows us to conclude that the value of a variable does not change during an interval if it does not change at any point in that interval. Instantiating the lemma with the predicates $\leq$ and $\geq$ is not as useful, but still needed in some situations.

3.3. Theorem Let $\mathtt{w}$ be a while-statement, let $\mathtt{v}$ be a mutable program variable, and let $\trianglelefteq$ be a reflexive and transitive predicate (e.g. $\simeq$, $\leq$ or $\geq$). Then the Value-Evolution trace lemma for $\mathtt{w}, \mathtt{v}$ and $\trianglelefteq$ is valid in $\mathbb{D} \cup \mathbb{I}$.

Proof. To simplify the presentation, we only prove the lemma for integer program variables. Adapting the proof to array program variables is straightforward. Let σ be an arbitrary grounding of both the enclosing iterations of $\mathtt{w}$ and of it_L and it_R, such that the instantiation of the premise of the Value-Evolution trace lemma with σ holds. Consider now the instance

$$\text{BC: } \sigma(v(tp_{\mathtt{w}}(it_L)) \trianglelefteq v(tp_{\mathtt{w}}(it_L))) \tag{3.8a}$$

$$\text{IC: } \sigma \forall it^{\mathbb{D}}. \Big((it_L \leq it < it_R \wedge v(tp_{\mathtt{w}}(it_L)) \trianglelefteq v(tp_{\mathtt{w}}(it))) \rightarrow v(tp_{\mathtt{w}}(it_L)) \trianglelefteq v(tp_{\mathtt{w}}(\mathtt{succ}(it))) \Big) \tag{3.8b}$$

$$\text{Con: } \sigma \forall it^{\mathbb{D}}. \Big(it_L \leq it \leq it_R \rightarrow v(tp_{\mathtt{w}}(it_L)) \trianglelefteq v(tp_{\mathtt{w}}(it)) \Big), \tag{3.8c}$$

of the generalized bounded induction axiom scheme with $(it) := v(tp_{\mathtt{w}}(it_L)) \trianglelefteq v(tp_{\mathtt{w}}(it))$, where additionally the grounding σ is applied. The base case (3.8a) holds since $\trianglelefteq$ is reflexive, and the inductive case (3.8b) holds since it is equal to the grounded premise of the Value-Evolution lemma. We therefore know that the conclusion (3.8c) holds. We next instantiate it in the conclusion (3.8c) to σit_R, which yields $\sigma(it_L \leq it_R \leq it_R \rightarrow v(tp_{\mathtt{w}}(it_L)) \trianglelefteq v(tp_{\mathtt{w}}(it_R)))$. Since $\leq$ is reflexive, we know that $\sigma(it_R \leq it_R)$ holds, from which we conclude $\sigma(it_L \leq it_R \rightarrow v(tp_{\mathtt{w}}(it_L)) \trianglelefteq v(tp_{\mathtt{w}}(it_R)))$. □

Dense program variables. The following two lemmas apply to integer program variables, which act as iterators. Such program variables are *dense*, defined as follows: Let

w be a while-statement, let v be a mutable program variable and define

$$Dense_{w,v} := \forall it^{\mathbb{D}}.\Big(it < lastIt_{\mathtt{w}} \rightarrow \\ \big(v(tp_{\mathtt{w}}(\mathtt{succ}(it))) = v(tp_{\mathtt{w}}(it)) \vee v(tp_{\mathtt{w}}(\mathtt{succ}(it))) = v(tp_{\mathtt{w}}(it)) + 1\big)\Big)$$

In our work, we assume that no array program variables are used as iterators. It would be straightforward, though, to generalize our approach to cover this corner case too.

Intermediate-Value trace lemma. Let w be a while-statement and v be a mutable integer program variable. Then the *Intermediate-Value trace lemma for* w *and* v is

$$\forall enclIts.\forall x^{\mathbb{I}}.\Big(\big(Dense_{w,v} \wedge v(tp_{\mathtt{w}}(\mathtt{0})) \leq x \wedge x < v(tp_{\mathtt{w}}(lastIt_{\mathtt{w}}))\big) \\ \rightarrow \exists it^{\mathbb{D}}.(it < lastIt_{\mathtt{w}} \wedge v(tp_{\mathtt{w}}(it)) \simeq x \wedge v(tp_{\mathtt{w}}(\mathtt{succ}(it))) \simeq v(tp_{\mathtt{w}}(it)) + 1)\Big)$$

The Intermediate-Value lemma lets us conclude that if the (iterator-) 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 a loop iteration it_0, in which v has exactly the value x and gets incremented. The value x usually denotes some position in an array. One can see this lemma as a discrete version of the Intermediate-Value Theorem for continuous functions.

3.4. Theorem Let w be a while-statement and let v be a mutable program variable. Then the Intermediate-Value trace lemma for w and v is valid in $\mathbb{D} \cup \mathbb{I}$.

Proof. We prove the following equivalent formula obtained from the Intermediate-Value lemma by modus tollens.

$$\forall enclIts.\forall x^{\mathbb{I}}.\Big(\Big(Dense_{w,v} \wedge v(tp_{\mathtt{w}}(\mathtt{0})) \leq x \wedge \\ \forall it^{\mathbb{D}}.\big((it < lastIt_{\mathtt{w}} \wedge v(tp_{\mathtt{w}}(\mathtt{succ}(it))) \simeq v(tp_{\mathtt{w}}(it)) + 1) \rightarrow v(tp_{\mathtt{w}}(it)) \not\simeq x\big)\Big) \\ \rightarrow v(tp_{\mathtt{w}}(lastIt_{\mathtt{w}})) \leq x\Big) \tag{3.9}$$

Let σ be an arbitrary grounding of both the enclosing iterations of w and of x, such that the instantiation of the premise of (3.9) with σ holds. Consider now

$$\text{BC: } \sigma(v(tp_{\mathtt{w}}(\mathtt{0})) \leq x) \tag{3.10a}$$

$$\text{IC: } \sigma\forall it^{\mathbb{D}}.\Big(\big(\mathtt{0} \leq it < lastIt_{\mathtt{w}} \wedge v(tp_{\mathtt{w}}(it)) \leq x\big) \rightarrow v(tp_{\mathtt{w}}(\mathtt{succ}(it))) \leq x\Big) \tag{3.10b}$$

$$\text{Con: } \sigma\forall it^{\mathbb{D}}.\Big(\mathtt{0} \leq it \leq lastIt_{\mathtt{w}} \rightarrow v(tp_{\mathtt{w}}(it)) \leq x\Big), \tag{3.10c}$$

obtained from the instance of the generalized bounded induction axiom scheme with $IH(it) := v(tp_{\mathtt{w}}(it)) \leq x$ by instantiating it_L and it_R to $\mathtt{0}$ resp. $lastIt_{\mathtt{w}}$, and applying the grounding σ.

The base case (3.10a) holds, since it occurs as the second premise of (3.9) instantiated with σ. For the inductive case (3.10b), assume $\sigma(\mathtt{0} \leq it < lastIt_{\mathtt{w}})$ and $\sigma(v(tp_{\mathtt{w}}(it)) \leq x)$. We proceed with a case distinction (which covers all cases, since $\sigma Dense_{w,v}$ holds):

- Assume $\sigma(v(tp_{\mathtt{w}}(\mathtt{succ}(it))) = v(tp_{\mathtt{w}}(it)))$. Since we also assume $\sigma(v(tp_{\mathtt{w}}(it)) \leq x)$, we immediately get $\sigma(v(tp_{\mathtt{w}}(\mathtt{succ}(it))) \leq x)$.
- Assume $\sigma(v(tp_{\mathtt{w}}(\mathtt{succ}(it))) = v(tp_{\mathtt{w}}(it)) + 1)$. Combined with the assumption $\sigma(it < lastIt_{\mathtt{w}})$ and the third premise of (3.9) instantiated with σ, we get $\sigma(v(tp_{\mathtt{w}}(it)) \not\approx x)$, which combined with $\sigma(v(tp_{\mathtt{w}}(it)) \leq x)$ and the totality-axiom of $<$ for integers gives $\sigma(v(tp_{\mathtt{w}}(it)) < x)$. Finally, we combine this fact with $\sigma(v(tp_{\mathtt{w}}(\mathtt{succ}(it))) = v(tp_{\mathtt{w}}(it)) + 1)$ and the integer-theory-lemma $x < y \rightarrow x + 1 \leq y$ to derive $\sigma(v(tp_{\mathtt{w}}(\mathtt{succ}(it))) \leq x)$.

Since we have concluded $\sigma(v(tp_{\mathtt{w}}(\mathtt{succ}(it))) \leq x)$ for all cases, we conclude that the inductive case (3.10b) holds. We therefore know that also the conclusion (3.10c) holds. Since the theory axiom $\forall it^{\mathbb{D}}.\mathtt{0} \leq it$ holds, (3.10c) implies the conclusion of (3.9) instantiated with σ, which concludes the proof. □

Iteration-Injectivity trace lemma. Let $\mathtt{w}$ be a while-statement and $\mathtt{v}$ be a mutable integer program variable. Then the *Iteration-Injectivity trace lemma for* $\mathtt{w}$ *and* $\mathtt{v}$ is

$$\forall enclIts.\forall it_L^{\mathbb{D}}, it_R^{\mathbb{D}}.\Big(\\ (Dense_{w,v} \wedge v(tp_{\mathtt{w}}(\mathtt{succ}(it_L))) = v(tp_{\mathtt{w}}(it_L)) + 1 \wedge it_L < it_R \wedge it_R \leq lastIt_{\mathtt{w}}) \\ \rightarrow v(tp_{\mathtt{w}}(it_L)) \not\approx v(tp_{\mathtt{w}}(it_R))\Big)$$

Iterator variables often have the property that they visit each element of the data structure at most once. This lemma 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 is not changed after the first visit, and in particular, that its value at the end of the loop is the same as its value after the first visit.

3.5. Theorem Let $\mathtt{w}$ be a while-statement and let $\mathtt{v}$ be a mutable program variable. Then the Iteration-Injectivity trace lemma for $\mathtt{w}$ and $\mathtt{v}$ is valid in $\mathbb{D} \cup \mathbb{I}$.

Proof. Let σ be an arbitrary grounding of both the enclosing iterations of $\mathtt{w}$ and of it_L and it_R, such that the instantiation of the premise of the Iteration-Injectivity lemma

with σ holds. Consider now

$$\text{BC: } \sigma(v(tp_{\mathtt{w}}(it_L)) < v(tp_{\mathtt{w}}(\mathtt{succ}(it_L)))) \tag{3.11a}$$

$$\text{IC: } \sigma\forall it^{\mathbb{D}}.\Big((\mathtt{succ}(it_L) \leq it < lastIt_{\mathtt{w}} \wedge v(tp_{\mathtt{w}}(it_L)) < v(tp_{\mathtt{w}}(it))) \tag{3.11b}$$

$$\rightarrow v(tp_{\mathtt{w}}(it_L)) < v(tp_{\mathtt{w}}(\mathtt{succ}(it)))\Big)$$

$$\text{Con: } \sigma\forall it^{\mathbb{D}}.\Big(\mathtt{succ}(it_L) \leq it \leq lastIt_{\mathtt{w}} \rightarrow v(tp_{\mathtt{w}}(it_L)) < v(tp_{\mathtt{w}}(it))\Big), \tag{3.11c}$$

obtained from the instance of the generalized bounded induction axiom scheme with $IH(it) := v(tp_{\mathtt{w}}(it_L)) < v(tp_{\mathtt{w}}(it))$, by instantiating it_L and it_R to $\mathtt{succ}(it_L)$ resp. $lastIt_{\mathtt{w}}$, and applying σ.

Combining the instantiated second premise $\sigma(v(tp_{\mathtt{w}}(\mathtt{succ}(it_L))) = v(tp_{\mathtt{w}}(it_L)) + 1)$ of the lemma with the integer-theory-axiom $\forall x^{\mathbb{I}}.x < x + 1$ yields that (3.11a) holds. For the inductive case, we assume for arbitrary but fixed it that $\sigma(v(tp_{\mathtt{w}}(it_L)) < v(tp_{\mathtt{w}}(it)))$ holds. Combined with $\sigma Dense_{w,v}$ and the integer theory axiom $\forall x^{\mathbb{I}}.(x < y \rightarrow x < y+1)$ this yields $\sigma(v(tp_{\mathtt{w}}(it_L)) < v(tp_{\mathtt{w}}(\mathtt{succ}(it))))$, so (3.11b) holds. Since both premises (3.11a) and (3.11b) hold, also the conclusion (3.11c) holds. Next, $\sigma(it_L < it_R)$ implies $\sigma(\mathtt{succ}(it_L) \leq it_R)$ (using the monotonicity of $\mathtt{succ}$). We therefore know $\sigma(\mathtt{succ}(it_L) \leq it_R < lastIt_{\mathtt{w}})$, and instantiate the conclusion (3.11c) to obtain $\sigma(v(tp_{\mathtt{w}}(it_L)) < v(tp_{\mathtt{w}}(it_R)))$. Finally, we use the integer theory axiom $\forall x^{\mathbb{I}}, y^{\mathbb{I}}.(x < y \rightarrow x \not\simeq y)$ to conclude $\sigma(v(tp_{\mathtt{w}}(it_L)) \not\simeq v(tp_{\mathtt{w}}(it_R)))$. □

At-Least-One-Iteration trace lemma. Let w be a while-statement `while(Cond){c}`. Then the *At-Least-One-Iteration trace lemma for* w is

$$\forall enclIts_w.([\![\mathtt{Cond}]\!](tp_{\mathtt{w}}(0)) \rightarrow \exists it^{\mathbb{D}}.\mathtt{succ}(it) \simeq lastIt_w)$$

In contrast to the other trace lemmas, this lemma does not cover any inductive consequence that we could not conclude from other axioms. It is still useful for theorem provers, which do not propagate disequality eagerly. Note that superposition-based provers do not conclude $0 \not\simeq \sigma lastIt_w$ from $[\![\mathtt{Cond}]\!](\sigma tp_{\mathtt{w}}(0))$ and $\neg[\![\mathtt{Cond}]\!](\sigma tp_{\mathtt{w}}(lastIt_{\mathtt{w}}))$, due to the used inference system.

3.6. Theorem Let $\mathtt{p}_0$ be a program with a while-statement $\mathtt{w} := \mathtt{while(Cond)\{c\}}$. Then the At-Least-One-Iteration trace lemma for w follows from $[\![\mathtt{p}_0]\!]$ in $\mathbb{D} \cup \mathbb{I}$.

Proof. Let σ be an arbitrary grounding of the enclosing iterations of w, such that the premise of the At-least-One-Iteration lemma holds. From $\sigma[\![\mathtt{Cond}]\!](\sigma tp_{\mathtt{w}}(0))$ and $\neg[\![\mathtt{Cond}]\!](\sigma tp_{\mathtt{w}}(lastIt_{\mathtt{w}}))$ we conclude $0 \not\simeq \sigma lastIt_w$. We then use the difference logic axiom $\forall it_1^{\mathbb{D}}.(0 \not\simeq it_1 \rightarrow \exists it_2^{\mathbb{D}}.\mathtt{succ}(it_2) = it_1)$ to conclude $\exists it^{\mathbb{D}}.\mathtt{succ}(it) \simeq \sigma lastIt_w$. □

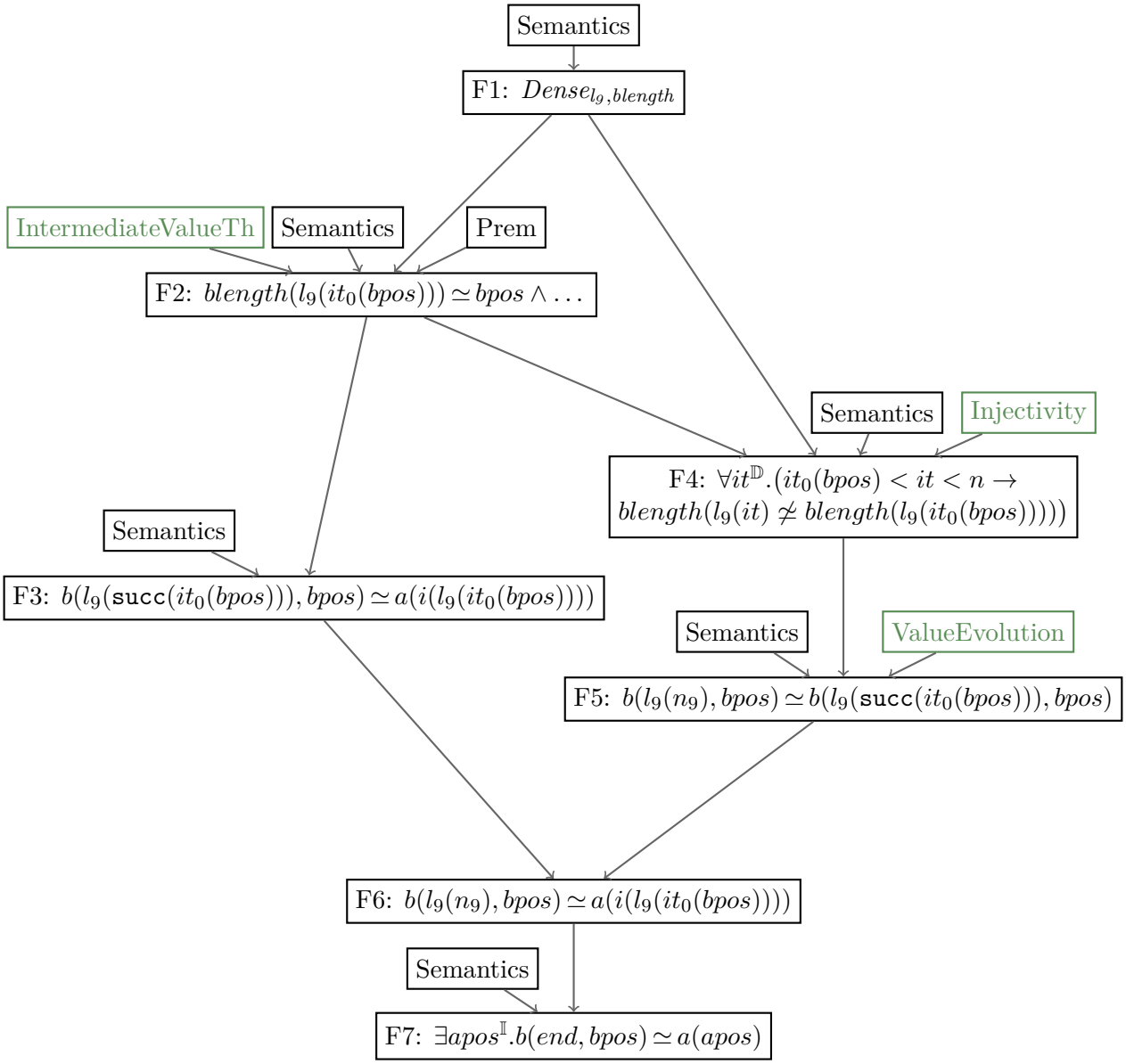

Figure 3.2: High-level proof sketch for the property of the running example.

3.3 A Correctness Proof for the Running Example

In Section 3.1, we discussed how to informally prove property (2.1) of the running example. In this section, we will show how to prove this property formally, using the verification framework introduced in Section 3.2.

An outline of the formal proof is visualized in Figure 3.2. Our presentation focuses on the important steps of the proof (that is, formulas F1-F7) and the trace lemmas needed

in the proof (that is, the formulas abbreviated as IntermediateValueTh, Injectivity, and ValueEvolution), and highlights the steps in the proof, which use (some part of) the program semantics.

The proof proceeds as follows. We start from the semantics of the program and from the trace lemmas, and additionally assume, for some nullary (skolem) function *bpos*, the premise of property (2.1), that is, Prem := $0 \leq bpos < blength$. First, we use the program semantics to derive formula F1, which denotes that `blength` is dense during the execution of the loop l_9. We then use the Intermediate-Value trace lemma for the loop at location l_9 and the variable `blength`. We combine it with F1, the semantics, and Prem, to conclude Formula F2, which in its full form is

$$\begin{aligned} &blength(l_9(it_0(bpos))) \simeq bpos \wedge \\ &it_0(bpos) < n_9 \wedge \\ &blength(l_9(\texttt{succ}(it_0(bpos)))) \simeq blength(l_9(it_0(bpos))) + 1. \end{aligned}$$

The formula F2 denotes that for each position *bpos* in `b`, there exists a loop iteration $it_0(bpos)$, such that (i) we visit *bpos* in iteration $it_0(bpos)$, (ii) the iteration $it_0(bpos)$ happens before the last loop iteration n_9, and (iii) we increment `blength` by `1` in that iteration $it_0(bpos)$. Note that the application of the Intermediate-Value trace lemma effectively synthesizes the term $it_0(bpos)$, as discussed in Subsection 3.1.2. Moreover, the term $it_0(bpos)$ represents a quantifier alternation, as it encodes an existentially quantified variable which depends on the universally quantified variable *bpos*[1].

Next, we combine Formula F2 with the semantics to derive Formula F3. The latter formula denotes that in the iteration $\texttt{succ}(it_0(bpos))$ following the iteration $it_0(bpos)$, the value of `b` at position *bpos* is equal to the value of `a` at the position corresponding to the value of `i` in the iteration $it_0(bpos)$. Note that the previously synthesized term $it_0(bpos)$ occurs in F3 as a subterm of the term $\texttt{succ}(it_0(bpos))$. It would be very hard to guess the importance of the latter term from the syntax of the program and the property, as it contains both a quantifier alternation and the successor symbol of the background theory $\mathbb{D}$.

Then, we use the Iteration-Injectivity trace lemma for l_9 and `blength`, by combining it with F1, F2, and the semantics. As a result, we obtain formula F4, which intuitively denotes that after iteration $it_0(bpos)$, the position *bpos* is not visited again by `blength`. Afterwards, we combine the Value-Evolution trace lemma for l_9 and `b` with F4 and the semantics. This yields the formula F5, which denotes that the value of `b` at position *bpos* at the end of the loop n_9 is the same as the value of `b` at position *bpos* at iteration $\texttt{succ}(it_0(bpos))$. Note that we instantiated the bounds it_1, it_2 of the Value-Evolution trace lemma with $\texttt{succ}(it_0(bpos))$ and n_9 as part of applying the trace lemma to derive the formula F4. These instantiations exemplify the specialization of a generalized bounded induction axiom to a part of the loop, as described in Subsection 3.1.3.

[1] While *bpos* occurs as a universally quantified variable in the conjecture, it is transformed into an existentially quantified variable as part of negating the conjecture, and is therefore encoded by a skolem function, which we here also refer to as *bpos*.

Finally, we combine formulas F3 and F5 to conclude formula F6, from which we immediately conclude F7. The latter formula corresponds to the conclusion of property (2.1) and denotes that there exists a position *apos*, such that the value of b at position *bpos* at the end of the loop is equal to the value of a at that position *apos*.

3.4 Related Work

Most verification approaches use a reasoning language without an explicit notion of timepoints to express programs and properties, and use invariants to establish program correctness [BGMR15, CC92]. Such invariants correspond to a fragment of trace logic restricted to formulas of the form $\forall it^{\mathbb{D}}.P(it)$, where P contains no existentially quantified variable of sort $\mathbb{D}$ and the only universally quantified variable of sort $\mathbb{D}$ contained in P is it. The lack of existential, and thus alternating, quantification makes this fragment suitable for automation via SMT-solving [BDW18, HB12, FQ02, ALGC12, DHKR11, KGC16], for programs where full first-order logic is not needed, for instance programs involving mainly integer variables and function calls. For program properties expressed in full first-order logic, such as over unbounded arrays, existing methods [GSV18, FPMG19, HR18, DDA10, CCL11, KFG20, CGU20, AGS14, DHK16, RL17] are not able yet to automatically verify program correctness. We argue that the missing expressiveness is the problem here. In particular, invariant-based languages are not able to express generalized bounded induction axioms, as those axioms include existentially quantified variables of sort $\mathbb{D}$ and multiple universally quantified variables of sort $\mathbb{D}$. As a result, these languages are not able to synthesize timepoints for loop splitting, as discussed in Subsection 3.1.2, and they are not able to specialize general trace lemmas to parts of the loop, as discussed in Subsection 3.1.3. In contrast, trace logic can express generalized bounded induction axioms and supports the automated loop splitting highlighted in Subsections 3.1.2 and 3.1.3.

Our approach to automate induction using trace lemmas is related to template-based invariant generation methods [CSS03, GR09]. Our trace lemmas are however more expressive than existing templates, as they contain existentially quantified variables of sort $\mathbb{D}$ and multiple universally quantified variables of sort $\mathbb{D}$. In particular, we use trace lemmas to enable automated loop splitting, which is not supported by template-based methods.

Recent efforts in using first-order theorem provers for proving software properties [KV09a, GKR18] are closely related to our work. While [KV09a, GKR18] only handle simple loops, our work supports a standard while-language with explicit locations and arbitrary nestings of statements. Furthermore, we prove the soundness and the Hoare-completeness of our reasoning language, and introduce and prove trace lemmas to automate inductive reasoning based on bounded induction over loop iterations. Finally, we are able to automate the verification of properties with arbitrary quantification, which could not be effectively achieved in [GKR18].

First-order reasoning for program analysis is also addressed in [BS01, BB13], by introducing dynamic trace logic, an extension of dynamic logic with modalities for reasoning about traces. A custom sequent calculus is proposed in [BS01, BB13], implying that automating the work would require the design of specialized sequent calculus provers. Unlike [BS01, BB13], our work is fully automated. Further, our work preserves the control-flow structure of programs by introducing function symbols and automates inductive reasoning using trace lemmas.

Several theorem provers implement basic techniques for inductive reasoning [RV19, Cru17, RK15], by applying instances of the standard induction axiom scheme during proof-search. None of these provers can provide the inductive reasoning needed for the trace logic domain. In particular, it is far beyond the capabilities of these provers to efficiently identify the instances of the generalized bounded induction axiom scheme needed to prove most examples from the trace logic domain.

CHAPTER 4

Relational Trace Logic

Many interesting properties from security applications can be expressed as properties over multiple computation traces. In this chapter, we will generalize trace logic, introduced in Chapter 2, and the trace-logic-based verification approach, introduced in Chapter 3, to handle such properties.

4.1 Extending Trace Logic to Multiple Traces

4.1.1 Extending the Language

In Section 2.2, we introduced the trace logic language $\mathcal{L}$, consisting of symbols S_{Tp}, which denote program locations in the program tree, and symbols S_n, S_V, S_R, which denote for an arbitrary execution the last loop iterations, values of program variables, and reachable timepoints. In this subsection, we generalize the trace logic language $\mathcal{L}$ to the relational trace logic language $\mathcal{L}^k$ (parameterized by the number of traces k), by introducing additional symbols to denote execution traces, and by extending the symbols in S_n, S_V, and S_R to explicitly state the trace they apply to. As a result, we will be able to use these symbols to describe values of multiple execution traces separately. Note that we do not need to generalize the symbols in S_{Tp} describing program locations, since these locations are independent of execution traces.

We use the following symbols in the relational trace logic $\mathcal{L}^k$. First, let $\mathbb{T}$ be an uninterpreted sort denoting computation traces, let $t_1, \ldots, t_k$ be nullary function symbols of sort $\mathbb{T}$ denoting k computation traces, and let S^k_{Tr} denote the set of all these function symbols t_i. Secondly, we reuse the existing symbols S_{Tp}, and the macros $tp_{\mathrm{s}}, tp_{\mathrm{s}}(it), start_{\mathrm{p}}, end_{\mathrm{p}}$ and *start* defined on top of the symbols S_{Tp}. Thirdly, we extend each symbol in S_n, S_V and S_R with an additional argument of sort $\mathbb{T}$ (we add the argument as last argument of the symbol). This results in sets S'_n, S'_V and S'_R. Furthermore, we adapt the definition of the macro $lastIt_{\mathrm{w}}$, by adding again an additional argument of sort $\mathbb{T}$. In particular,

for any loop w and any argument t_i of sort $\mathbb{T}$ we define

$$lastIt_{\mathtt{w}}(t) := n_w(it^{w_1}, \ldots, it^{w_k}, t_i).$$

4.1. Example Recall the running example from Figure 2.1. As before, we use l_6 to refer to the timepoint corresponding to the first assignment of `blength` in the program, and use $l_9(0)$ to denote the timepoint corresponding to evaluating the loop condition in the first iteration. We now use $l_9(n_9(t_1))$ and $l_9(n_9(t_2))$ to denote the timepoint corresponding to evaluating the loop condition in the last loop iteration of the execution trace t_1 resp. t_2, and use $Reach(l_9(\mathtt{succ}(zero)), t_1)$ to denote that the execution trace t_1 reaches the timepoint $l_9(\mathtt{succ}(zero))$. Furthermore, we now use $\mathtt{blength}(l_8, t_2)$ to refer to the value of program variable `blength` in execution trace t_2 at the moment before `i` is first assigned, and write $b(l_{11}, i(l_{11}(it), t_1), t_1)$ for the value of array `b` in execution trace t_1 at timepoint $l_{11}(it)$ at position pos, where pos is the value of `i` in execution trace t_1 at timepoint $l_{11}(it)$. Finally, we now write $a(0, t_1)$ to denote the value of the array `a` in execution trace t_1 at position 0, and write $i(l_{end}, t_1) + 1$ to denote the value of expression `i+1` at the end of execution trace t_1. □

Finally, we define the signature $Sig(\mathcal{L}^k)$ of the generalized trace logic $\mathcal{L}^k$ as

$$Sig(\mathcal{L}^k) \;:=\; (S_{\mathbb{D}} \cup S_{\mathbb{I}}) \;\cup\; (S_{Tp} \cup S'_n \cup S'_R \cup S'_V) \;\cup\; S^k_{Tr},$$

and define *relational trace logic*, denoted by $\mathcal{L}^k$, as the instance of many-sorted first-order logic with equality modulo $\mathbb{D} \cup \mathbb{I}$ with signature $Sig(\mathcal{L}^k)$.

To simplify the presentation in the following subsections, we introduce the following macro. For an arbitrary program variable `v`, let $EqTr_v(tp)$ denote that `v` has the same value(s) in both traces t_i, t_j at timepoint tp, that is

$$EqTr_v^{t_i,t_j}(tp) := \begin{cases} \forall pos^{\mathbb{I}}.v(tp, pos, t_i) \simeq v(tp, pos, t_j)) & \text{if } v \text{ is mutable array} \\ \forall pos^{\mathbb{I}}.v(pos, t_i) \simeq v(pos, t_j)) & \text{if } v \text{ is constant array} \\ v(tp, t_i) \simeq v(tp, t_j)) & \text{if } v \text{ is mutable variable} \\ v(t_i) \simeq v(t_j) & \text{if } v \text{ is constant variable} \end{cases}$$

We will sometimes omit to denote the traces t_1, t_2 that $EqTr$ applies to, if they are clear from the context.

4.1.2 Extending the Axiomatic Semantics

We now generalize the axiomatic semantics, which we presented in Section 2.4 for the non-relational setting, to relational trace logic $\mathcal{L}^k$. Consider the transformation $\mathcal{R}$, which maps an arbitrary trace logic formula F and an arbitrary execution trace t_i to a relational trace logic formula $\mathcal{R}(F, t_i)$, by (i) replacing each symbol in F, which is contained in S_n, S_V, or S_R, with the corresponding symbol contained in S'_n, S'_V, or S'_R, and (ii) adding t_i as the last argument to all occurrences of the symbols contained in S'_n, S'_V, and S'_R.

```
1  func main()
2  {
3     Int x;
4     if(x > 0)
5     {
6        x = x - 1;
7     }
8     else
9     {
10       skip;
11    }
12 }
```

Figure 4.1: A simple program.

Given a program $\mathtt{p}_0$ and a trace t_i, we then define the axiomatic semantics of $\mathtt{p}_0$ with respect to t_i as

$$[\![\mathtt{p}_0, t_i]\!] := \mathcal{R}([\![\mathtt{p}_0]\!], t_i).$$

4.2. Example Consider the program $\mathtt{p}_0$ denoted in Figure 4.1. Recall from 2.4 that the axiomatic semantics of $\mathtt{p}_0$ is defined in trace logic $\mathcal{L}$ as

$$\begin{aligned}[\![\mathtt{p}_0]\!] := \quad & x(l_4) > 0 \rightarrow x(l_6) \simeq x(l_4) && \wedge \\ & x(l_4) > 0 \rightarrow x(l_{end}) \simeq x(l_6) - 1 && \wedge \\ & x(l_4) \not> 0 \rightarrow x(l_{10}) \simeq x(l_4) && \wedge \\ & x(l_4) \not> 0 \rightarrow x(l_{end}) \simeq x(l_{10})\end{aligned}$$

The axiomatic semantics of $\mathtt{p}_0$ with respect to the execution trace t_1 is defined in relational trace logic $\mathcal{L}^k$ as

$$\begin{aligned}[\![\mathtt{p}_0, t_1]\!] := \quad & x(l_4, t_1) > 0 \rightarrow x(l_6, t_1) \simeq x(l_4, t_1) && \wedge \\ & x(l_4, t_1) > 0 \rightarrow x(l_{end}, t_1) \simeq x(l_6, t_1) - 1 && \wedge \\ & x(l_4, t_1) \not> 0 \rightarrow x(l_{10}, t_1) \simeq x(l_4, t_1) && \wedge \\ & x(l_4, t_1) \not> 0 \rightarrow x(l_{end}, t_1) \simeq x(l_{10}, t_1)\end{aligned}$$

□

We finally adapt the formalization of the partial correctness of a program property, given in Subsection 2.4.3, to the setting of hyperproperties as follows. Let $\mathtt{p}_0$ be a program, and let F be a property of $\mathtt{p}_0$ expressed in $\mathcal{L}^k$. Using $\mathcal{L}^k$, we formally capture the partial correctness of F with respect to $\mathtt{p}_0$ as the entailment

$$[\![\mathtt{p}_0, t_1]\!] \wedge \cdots \wedge [\![\mathtt{p}_0, t_k]\!] \vDash_{\mathbb{DUI}} F.$$

4.1.3 Extending the Verification Framework

Using relational trace logic, we establish the partial correctness of a relational property of a program using a validity check in first-order logic modulo $\mathbb{D} \cup \mathbb{I}$. Analogously to the non-relational setting, we can therefore use any off-the-shelf first-order theorem prover to reason about the partial correctness of relational properties, and can use any off-the-shelf first-order proof system supported by the theorem prover to certify the partial correctness of the given property. Similar to the non-relational setting, existing theorem provers are not able to handle the inductive reasoning required to reason about relational properties. To address this challenge, we (i) reuse the existing trace lemmas presented in Subsection 3.2.1, and (ii) add two additional trace lemmas specific to relational properties, as follows.

Relational-Equality-Preservation trace lemma. Let w be a while-statement, v be a mutable program variable, and let t_i and t_j be two different symbols denoting traces.

If v is an integer program variable, then the *Relational-Equality-Preservation trace lemma for w, v, t_i, and t_j* is the instance of the generalized bounded induction axiom scheme

$$\forall enclIts_w.\forall it_L^{\mathbb{D}}, it_R^{\mathbb{D}}.\Big(\Big(BC(it_L) \wedge IC(it_L, it_R)\Big) \rightarrow Concl(it_L, it_R)\Big),$$

where $BC(it_L), IC(it_L, it_R)$, and $Concl(it_L, it_R)$ are defined according to Formulas (2.8)-(2.10), for the induction hypothesis $IH(it) := v(tp_{\mathtt{w}}(it), t_i) \simeq v(tp_{\mathtt{w}}(it), t_j)$.

If v is an array program variable, then the *Relational-Equality-Preservation trace lemma for w, v, t_i, and t_j* is the instance of the generalized bounded induction axiom scheme

$$\forall enclIts_w.\forall pos^{\mathbb{I}}, it_L^{\mathbb{D}}, it_R^{\mathbb{D}}.\Big(\Big(BC(it_L) \wedge IC(it_L, it_R)\Big) \rightarrow Concl(it_L, it_R)\Big),$$

where $BC(it_L), IC(it_L, it_R)$, and $Concl(it_L, it_R)$ are defined according to Formulas (2.8)-(2.10), for the induction hypothesis $IH(it) := v(tp_{\mathtt{w}}(it), pos, t_i) \simeq v(tp_{\mathtt{w}}(it), pos, t_j)$.

This lemma is central for reasoning about relational properties, as it often allows us to conclude that a variable has the same value in traces t_i and t_j at a certain loop iteration, or even at all loop iterations.

4.3. Theorem Let w be a while-statement, let v be a mutable program variable, and let t_i and t_j be two different symbols denoting traces. Then the Relational-Equality-Preservation trace lemma for w, v, t_i, and t_j is valid in $\mathbb{D} \cup \mathbb{I}$.

Proof. The Relational-Equality-Preservation trace lemma is an instance of the generalized bounded induction axiom scheme and therefore valid in $\mathbb{D} \cup \mathbb{I}$. □

Simultaneous-Termination trace lemma. Let w be a while-statement `while(Cond){c}` and let t_i and t_j be two different symbols denoting traces. Let further V denote the set of function-symbols denoting program variables occurring in the loop condition of w. Finally, consider the macro

$$EqV(it) := \bigwedge_{v \in V} EqTr_v^{t_i,t_j}(tp_{\mathtt{w}}(it)). \tag{4.1}$$

Intuitively, $EqV(it)$ denotes that each variable occurring in the loop condition has the same value in the traces t_i and t_j in the iteration it of the loop w. Note that $EqV(it)$ is a sufficient condition to conclude that the loop condition check of w in iteration it evaluates to the same value in t_i and t_j.

Then the *Simultaneous-Termination trace lemma for* w, t_1 *and* t_2 is

$$\begin{aligned}\forall enclIts_w.\Big(\Big(&Reach(start_{\mathtt{w}}, t_i) \wedge Reach(start_{\mathtt{w}}, t_j) \wedge EqV(\mathtt{0}) \wedge \\ &\forall it^{\mathbb{D}}.((it < lastIt_w(t_i) \wedge it < lastIt_w(t_j) \wedge EqV(it)) \rightarrow EqV(\mathtt{succ}(it)))\Big) \\ &\rightarrow lastIt_w(t_i) \simeq lastIt_w(t_j)\Big)\end{aligned}$$

The Simultaneous-Termination trace lemma represents a sufficient condition to conclude that both loops terminate in the same number of iterations, which is usually the first step in a correctness proof for hyperproperties. The lemma is based on the observation that the variables used in the loop condition check often have the same value in both traces.

4.4. Theorem Let $\mathtt{p}_0$ be a program containing a while-statement $\mathtt{w} := $ `while(Cond){c}`, and let t_i and t_j be two different symbols denoting traces. Then the Simultaneous-Termination trace lemma for w, t_i, and t_j follows from $[\![\mathtt{p}_0]\!]$ in $\mathbb{D} \cup \mathbb{I}$.

Proof. First note that $it < lastIt_w(t_i) \wedge it < lastIt_w(t_j)$ is equivalent to

$$it < min(lastIt_w(t_i), lastIt_w(t_j)).$$

Let now σ be an arbitrary grounding of both the enclosing iterations of w, such that the instantiation of the premise of the Simultaneous-Termination trace lemma with σ holds. Consider

$$\text{BC: } \sigma EqV(\mathtt{0}) \tag{4.2a}$$

$$\text{IC: } \sigma\forall it^{\mathbb{D}}.\Big((\mathtt{0} \leq it < min(lastIt_w(t_i), lastIt_w(t_j)) \wedge EqV(it)) \rightarrow EqV(\mathtt{succ}(it))\Big) \tag{4.2b}$$

$$\text{Con: } \sigma\forall it^{\mathbb{D}}.\Big(\mathtt{0} \leq it \leq min(lastIt_w(t_i), lastIt_w(t_j)) \rightarrow EqV(it)\Big), \tag{4.2c}$$

obtained from the instance of the induction axiom scheme with $IH(it) := EqV(it)$ by instantiating it_L and it_R to $\mathtt{0}$ resp. $min(lastIt_w(t_i), lastIt_w(t_j))$, and applying σ. Both the base case (4.2a) and the inductive case (4.2b) hold, since they occur as part of the instantiated premise of the lemma. Therefore also the conclusion (4.2c) holds, and in particular $\sigma EqV(min(lastIt_w(t_i), lastIt_w(t_j)))$ holds. Using the definition of EqV, we then derive

$$\sigma[\![\mathtt{Cond}]\!](min(lastIt_w(t_i), lastIt_w(t_j)), t_i) \leftrightarrow$$
$$\sigma[\![\mathtt{Cond}]\!](min(lastIt_w(t_i), lastIt_w(t_j)), t_j).$$

Next we show that neither $\sigma(lastIt_w(t_i) < lastIt_w(t_j))$ nor $\sigma(lastIt_w(t_j) < lastIt_w(t_i))$ holds.

- Assume that $\sigma(lastIt_w(t_i) < lastIt_w(t_j))$ holds: Since $\sigma Reach(start_{\mathtt{w}}, t_i)$ holds, we conclude that $[\![\mathtt{w}, t_i]\!]$ holds. In particular the fact $\sigma\neg[\![\mathtt{Cond}]\!](lastIt_w(t_i), t_i)$ holds (formula (2.19b)), which we rewrite to $\sigma\neg[\![\mathtt{Cond}]\!](lastIt_w(t_i), t_j)$. The latter fact contradicts $\sigma[\![\mathtt{Cond}]\!](lastIt_w(t_i), t_j)$, which follows from the semantics $[\![\mathtt{w}, t_i]\!]$, by instantiating (2.19a) with σ and combining the result with the assumption $\sigma(lastIt_w(t_i) < lastIt_w(t_j))$.
- Assume that $\sigma(lastIt_w(t_j) < lastIt_w(t_i))$ holds: Analogously to the previous case.

By the totality of $<$, we therefore know that $\sigma(lastIt_w(t_i) = lastIt_w(t_j))$ holds, which concludes the proof. □

4.2 Security Properties in Relational Trace Logic

In this section, we showcase the expressiveness of trace logic $\mathcal{L}^k$ by formalizing the two fundamental security properties non-interference [SM03] and sensitivity [DMNS06] in it.

4.2.1 Non-Interference

For many application scenarios, data can be partitioned into private and public data. In such a context, we are often interested in programs, which preserve the privacy of the private data, but at the same time interact with the environment using the public data. Non-interference [GM82] is a security property, which intuitively ensures that no information flow from private data to publically observable data occurs while executing a given program. In other words, non-interference ensures that publically observable data is independent from private data, that is, it only depends on the publically observable data itself.

We capture non-interference as a property in the relational trace logic $\mathcal{L}^2$, as follows. Let $\mathtt{p}_0$ be a program, let H be a set of high confidentiality variables to capture private data and confidential variables, and let L be a set of low confidentiality variables to capture publically observable data, variables and outputs. Recall further from Section 2.2 that *start* and l_{end} denote the first resp. last timepoint of the execution.

```
1  func main()
2  {
3      const Int hi;
4      Int low;
5
6      if(hi > 0)
7      {
8          low = low + 1;
9      }
10     else
11     {
12         low = low + 1;
13     }
14 }
```

(a) Branching on *high confid.* variable.

```
1  func main()
2  {
3      Int[] output;
4      const Int length;
5      const Int low;
6      Int hi = low;
7
8      Int i = 0;
9      while(hi < length)
10     {
11         output[i] = hi;
12         hi = hi + 1;
13         i = i + 1;
14     }
15 }
```

(b) Explicit flow.

Figure 4.2: Examples with non-interference behavior.

Non-interference then expresses that given the same values for all low confidentiality variables L at the beginning *start* of two arbitrary execution traces t_1, t_2, the values of all low confidentiality variables L at the end l_{end} of these execution traces are the same.

$$(\bigwedge_{v \in L} EqTr_v(start)) \rightarrow (\bigwedge_{v \in L} EqTr_v(l_{end})). \tag{4.3}$$

4.5. Example Consider the program illustrated in Figure 4.2a, containing a high confidentiality variable $\texttt{hi} \in H$, and a low confidentiality variable $\texttt{low} \in L$. For this program, non-interference, defined in (4.3), corresponds to the property

$$EqTr_{low}(start) \rightarrow EqTr_{low}(l_{end}). \tag{4.4}$$

This example is interesting, since in general a high confidentiality variable occurring in a branching condition of an if-then-else-statement, could potentially prevent non-interference. However, in the given program, the variable `low` is updated in the same way in both branches. As a result, non-interference still holds for the given program. □

4.6. Example Consider the program presented in Figure 4.2b, which contains a high confidentiality variable `hi` and the low confidentiality variables `length`, `output`, `i` and `low`. The program iteratively leaks values to the publically observable variable `output`. For the given program, non-interference corresponds to the property

$$\begin{gathered}(EqTr_{length}(start) \wedge EqTr_{low}(start) \wedge EqTr_{output}(start)) \\ \rightarrow \\ EqTr_{output}(l_{end})\end{gathered} \tag{4.5}$$

This example is interesting, as an explicit flow from the high confidentiality variable `hi` to the low confidentiality variable `output` occurs. The program nonetheless fulfills non-interference, as `hi` was already assigned the value of the low confidentiality variable `low` at line 5, before executing the loop at line 9. □

4.2.2 Sensitivity

Sensitivity is a property describing how much a program maximally amplifies differences in its inputs. It is part of the core of the Laplace mechanism used to enforce differential privacy [DMNS06]. For simplicity, we investigate the special case where (i) the difference in the input is limited to a single variable, (ii) the output consists of a single variable, and (iii) the program is required not to amplify the differences in the input at all.

For this special case, we capture sensitivity as a property in the relational trace logic $\mathcal{L}^2$, as follows. First, consider the macro $\mathit{Diff}_v(tp, z)$. For an integer program variable $\mathtt{v}$, it denotes that $\mathtt{v}$ differs by at most z between traces t_1, t_2 at timepoint tp. Formally,

$$\mathit{Diff}_v(tp, z) := |v(tp, t_1) - v(tp, t_2)| < z.$$

For an array program variable $\mathtt{v}$, it denotes that $\mathtt{v}$ differs at some position by at most z between traces t_1, t_2 at timepoint tp and has the same value in traces t_1, t_2 for all other positions at timepoint tp, that is,

$$\begin{aligned}\mathit{Diff}_v(tp, z) := \quad & \exists pos_1^{\mathbb{I}}.\Big(|v(tp, pos_1, t_1) - v(tp, pos_1, t_2)| < z \ \wedge \\ & \forall pos_2^{\mathbb{I}}.(pos_1 \not\simeq pos_2 \rightarrow v(tp, pos_2, t_1) \simeq v(tp, pos_2, t_2))\Big).\end{aligned}$$

Secondly, let $\mathtt{p_0}$ be a program, let V denote the variables occurring in $\mathtt{p_0}$, and let t_1 and t_2 denote two execution traces. Let further $\mathtt{v}$ be a variable of $\mathtt{p_0}$. Let finally $\mathtt{v'}$ be an integer variable, which intuitively denotes the output of $\mathtt{p_0}$.

Then the *sensitivity* of $\mathtt{p_0}$ with respect to $\mathtt{v}$ is defined as

$$\forall z^{\mathbb{I}}.\Big((\mathit{Diff}_v(start, z) \wedge \bigwedge_{v \in V \setminus \{v\}} \mathit{EqTr}_v(start)) \rightarrow \mathit{Diff}_{v'}(l_{end}, z)\Big) \tag{4.6}$$

4.7. Example In Figure 4.3, the contents of an array `a` are summed up into a variable `x`. For this program, sensitivity corresponds to the trace logic property

$$\forall z^{\mathbb{I}}.\ \Big((\mathit{Diff}_y(start, z) \wedge \mathit{EqTr}_a(start) \wedge \mathit{EqTr}_{alength}(start)) \rightarrow \mathit{Diff}_x(l_{end}, z)\Big) \tag{4.7}$$

stating that if the values of the variable `y` differ by at most z between two traces while all other array elements are equal, then the final values of `x` in these two traces will differ from each other by at most z as well. □

```
1   func main()
2   {
3       const Int[] a;
4       const Int alength;
5       const Int y;
6       Int x = 0;
7
8       Int i = 0;
9       while(i < alength)
10      {
11          x = x + a[i];
12          i = i + 1;
13      }
14      x = x + y;
15  }
```

Figure 4.3: Example adhering sensitivity.

4.3 Related Work

Verification of relational- and hyperproperties is an active area of research, with applications in programming languages and compilers, security, and privacy; see [BU18] for an overview. Various static analysis techniques have been proposed to analyze non-interference, such as type systems [SM03] and graph dependency analysis [GHM13]. Type systems proved also effective in the verification of privacy properties for cryptographic protocols [EM13, BFG+14, CEK+15, CGLM17, CGLM18]. Relational Hoare logic was introduced in [Ben04] and further extended in [BCK11, BCK13] for defining product programs to reduce relational verification to standard verification. All these works closely tie verification to the syntactic program structure, thus limiting their applicability and expressiveness. For instance, the existing syntax-driven, non-interference verification techniques from [SM03, GHM13] would consider the examples presented in Figure 4.2a and Figure 4.2b insecure. In contrast, our value-sensitive approach proves these examples to be secure. Recently, [GMF+18] encodes relational properties through refinement types in F* [SHK+16]. While still being syntax driven, [GMF+18] can potentially verify semantic properties by using SMT solving, although this typically requires the manual insertion and proof of program-dependent lemmas, which is not the case for us.

In [GS13] bounded model checking is proposed for program equivalence. Program equivalence is reduced in [FGK+14] to proving a set of Horn clauses, by combining a relational weakest precondition calculus with SMT-based reasoning. However, when addressing programs with different control flow as in [FGK+14], user guidance is required for proving program equivalence. Program equivalence is also studied in [ZHH17, KHE17] for proving information flow properties. Unlike these works, we are not limited to SMT solving and target the verification of relational properties expressed in full first-order

theories, possibly with alternations of quantifiers.

Motivated by applications to translation validation, the work of [NS16] develops powerful techniques for proving the correctness of loop transformations. Relational methods for reasoning about program versions and semantic differences are also introduced in [PY14, LHKR12]. Going beyond relational properties, an SMT-based framework for verifying k-safety properties is introduced in [SD16] and further extended [SDL18] for proving the correctness of 3-way merge. While these works focus on high-level languages, many others consider low-level languages, see [SD08, STL11, SSCA13, BDG14] for some exemplary approaches. Further afield, several authors have introduced logics for modeling hyperproperties. Unlike these works, trace logic allows expressing first-order relational properties and targets reasoning about such properties using arbitrary off-the-shelf first-order theorem provers, overcoming thus the SMT-based limitations of quantified reasoning.

Finally, in [CFK+14] HyperLTL and HyperCTL* is introduced to model temporal and relational properties. However, these logics support only decidable fragments of first-order logic and thus cannot handle relational properties with non-constant function symbols. As such, security and privacy properties over unbounded data structures/uninterpreted functions cannot be encoded or verified.

CHAPTER 5

Reasoning in Trace Logic using Vampire

The previous chapters 2, 3, and 4 showed how to formulate the correctness of software program properties as validity statements in standard multi-sorted first-order logic with equality modulo the background theory $\mathbb{D} \cup \mathbb{I}$. In the following chapters 5, 6, 7, 8, and 9, we turn our attention to reasoning about the resulting formalizations using superposition-based theorem proving, and in particular the state-of-the-art superposition-based theorem prover Vampire.

We start this chapter by recalling the main ideas from superposition-based theorem proving. Afterwards, we describe the concrete design used in Vampire, including the options surfaced to the user for controlling proof search. Finally, we discuss how to tune Vampire to the trace logic domain by leveraging the existing user options.

5.1 Background on Saturation-Based Theorem Proving

5.1.1 Automated Theorem Proving

We consider standard many-sorted first-order logic with equality modulo $\mathbb{D} \cup \mathbb{I}$, as described in Section 2.1, and recall additional standard definitions used in automated theorem proving.

A *literal* is an atomic formula or its negation. A literal $s \simeq t$ is called an *equality literal*. A *clause* is a (not necessarily binary) disjunction of literals. We often consider clauses as multisets of literals and denote by $\subseteq_M$ the subset relation among multisets. The *empty clause*, denoted by $\bot$, is the nullary disjunction of literals, and always evaluates to false. A clause that only consists of one equality literal is called a *unit equality clause*.

An *expression* E is a term, literal, or clause. We write $E[s]$ to mean an expression E with a particular occurrence of a term s. A *substitution*, denoted by σ, is any finite mapping of the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $n > 0$. Applying a substitution σ to an expression E yields another expression, denoted by $E\sigma$, by simultaneously replacing each x_i by t_i in E. We say that $E\sigma$ is an *instance* of E. For two substitutions σ_1 and σ_2, we say that σ_1 is *more general* than σ_2, if there exists a substitution σ_3, such that the composition of σ_1 and σ_3 yields σ_2. A *unifier* of two expressions E_1 and E_2 is a substitution σ such that $E_1\sigma = E_2\sigma$. If two expressions have a unifier, they also have a *most general unifier (mgu)*. The most general unifier is interesting, as it minimizes the amount of instantiation necessary to unify two terms. A *match* of expression E_1 to expression E_2 is a substitution σ such that $E_1\sigma = E_2$. Note that any match is a unifier (assuming the sets of variables in E_1 and E_2 are disjoint), but not vice-versa, as illustrated below.

5.1. Example Let E_1 and E_2 be the clauses $Q(x, y) \vee R(x, y)$ and $Q(c, d) \vee R(c, z)$, respectively. The only possible match of $Q(x, y)$ to $Q(c, d)$ is $\sigma_1 = \{x \mapsto c, y \mapsto d\}$. On the other hand, the only possible match of $R(x, y)$ to $R(c, z)$ is $\sigma_2 = \{x \mapsto c, y \mapsto z\}$. As σ_1 and σ_2 are not the same, there is no match of E_1 to E_2. Note however that E_1 and E_2 can be unified; for example, using $\sigma_3 = \{x \mapsto c, y \mapsto d, z \mapsto d\}$. □

5.1.2 Superposition Inference System

We assume basic knowledge in first-order theorem proving and superposition reasoning [BGML01, NR01]. We adopt the notations and the inference system of superposition from [KV13]. We recall that first-order provers perform inferences on formulas[1] using inference rules, where an *inference* is usually written as:

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

with $n \geq 0$. The formulas $F_1, \ldots, F_n$ are called the premises and F is called the conclusion of the inference above. An inference rule is a set of (concrete) inferences and an inference system is a set of inference rules. Given an inference system $\mathcal{I}$, a *derivation* from axioms A is a finite acyclic directed graph (DAG), where (i) each node is a formula and (ii) each node either is an axiom in A and does not have any incoming edges, or is a formula $F \notin A$, such that the incoming edges of F are exactly $(F_1, F), \ldots, (F_n, F)$ and there exists an inference $(F_1, \ldots, F_n, F) \in I$, for some inference rule $I \in \mathcal{I}$. A *proof* of a formula F from axioms A is a derivation from axioms A which contains a node F. A *refutation* of axioms A is a proof of the empty clause $\bot$ from axioms A.

Let $\mathcal{T}$ be some fixed background theory. An inference is *sound* (for $\mathcal{T}$), if its conclusion is a logical consequence of its premises in first-order logic with equality (modulo $\mathcal{T}$). An inference system is *sound* (for $\mathcal{T}$) if all its inference rules are sound (for $\mathcal{T}$). As a

[1] During saturation, inferences are only performed on clauses.

consequence, if there exists a refutation of a set of clauses C in an inference system, which is sound (for $\mathcal{T}$), then C is unsatisfiable in first-order logic with equality (modulo $\mathcal{T}$). From now on, we will only consider inferences which are sound (for the theory we reason in). An inference system $\mathcal{I}$ is *complete* resp. *complete for* $\mathcal{T}$, if for any set C of clauses, which is unsatisfiable in first-order logic with equality resp. first-order logic with equality modulo $\mathcal{T}$, there exists a refutation from the clauses in C in $\mathcal{I}$.

Modern first-order theorem provers implement the *superposition inference system* for first-order logic with equality. This inference system is parameterized by a *simplification ordering* over terms and a *literal selection function* over clauses. In what follows, we denote by $\succ$ a simplification ordering over terms, that is, $\succ$ is a well-founded partial ordering satisfying the following three conditions:

- *stability under substitutions*: if $s \succ t$, then $s\theta \succ t\theta$;
- *monotonicity*: if $s \succ t$, then $l[s] \succ l[t]$;
- *subterm property*: $s \succ t$ whenever t is a proper subterm of s.

From now on, we only consider simplification orderings that are total on ground terms. The provided simplification ordering $\succ$ on terms is extended to a simplification ordering on literals, using a multiset extension of orderings, which ensures that (i) negative literals are always larger than their positive counterparts, (ii) if $L_1 \succ L_2$, where L_1 and L_2 are positive, then $\neg L_1 \succ L_1 \succ \neg L_2 \succ L_2$, and (iii) any equality literal is smaller than any literal using a predicate different than $\simeq$. The simplification ordering on literals is extended to a simplification ordering on clauses using another multiset extension. For simplicity, the extension of $\succ$ to literals and clauses will also be denoted by $\succ$. Whenever $E_1 \succ E_2$ for expressions E_1, E_2, we say that E_1 is larger than E_2 and E_2 is smaller than E_1 w.r.t. $\succ$. We say that an equality literal $s \simeq t$ is *oriented*, if $s \succ t$ or $t \succ s$.

A *literal selection function* selects at least one literal in every non-empty clause. In what follows, selected literals in clauses will be underlined: when writing $\underline{L} \vee C$, we mean that (at least) L is selected in $L \vee C$. In what follows, we assume that selection functions are *well-behaved* w.r.t. $\succ$: either a negative literal is selected or all maximal literals w.r.t. $\succ$ are selected [KV13].

In the sequel, we fix a simplification ordering $\succ$ and a well-behaved selection function and consider the superposition inference system, denoted by Sup, parametrized by these two ingredients. The inference system Sup for first-order logic with equality consists of the inference rules of Figure 5.1. It is both sound and complete (with respect to first-order logic with equality).

5.1.3 The Given-Clause Algorithm

We now overview the main ingredients in organizing proof search within first-order provers, using the superposition calculus. For details, we refer to [BGML01, NR01, KV13].

- Resolution and Factoring

$$\frac{\underline{L} \vee C_1 \qquad \underline{\neg L'} \vee C_2}{(C_1 \vee C_2)\sigma} \qquad\qquad \frac{\underline{L} \vee \underline{L'} \vee C}{(L \vee C)\sigma}$$

  where L is not an equality literal and $\sigma = mgu(L, L')$

- Superposition

$$\frac{\underline{s \simeq t} \vee C_1 \qquad \underline{L[s']} \vee C_2}{(C_1 \vee L[t] \vee C_2)\theta}$$

$$\frac{\underline{s \simeq t} \vee C_1 \qquad \underline{l[s'] \simeq l'} \vee C_2}{(C_1 \vee l[t] \simeq l' \vee C_2)\theta} \qquad \frac{\underline{s \simeq t} \vee C_1 \qquad \underline{l[s'] \not\simeq l'} \vee C_2}{(C_1 \vee l[t] \not\simeq l' \vee C_2)\theta}$$

  where s' not a variable, L is not an equality, $\theta = mgu(s, s')$, $t\theta \not\succ s\theta$ and $l'\theta \not\succ l[s']\theta$

- Equality Resolution and Equality Factoring

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

  where $\theta = mgu(s, s')$, $t\theta \not\succ s\theta$ and $t'\theta \not\succ t\theta$

Figure 5.1: The superposition calculus Sup.

Superposition-based provers use *saturation algorithms*: applying all possible inferences of Sup in a certain order to the clauses in the search space until (i) the empty clause has been derived, (ii) no more inferences can be applied, or (iii) a timeout is reached.

All state-of-the-art superposition-based theorem provers realize saturation using some variant of the *given-clause algorithm* [Vor01, Sch13]. The algorithm maintains two sets of clauses, referred to as *Active* resp. *Passive*. Intuitively, *Active* contains all clauses of the current proof attempt at which we can further extend the proof attempt, while *Passive* represents a one-step lookahead consisting of clauses, which can be derived from active clauses using a single inference step in the superposition calculus.[2] The given-clause algorithm now works as follows. Initially, *Passive* contains all input clauses and *Active* is empty. The algorithm then proceeds in rounds. In each round, a single clause C is selected from the one-step lookahead *Passive* using the so-called *clause selection heuristics*. The clause C is then moved to *Active* (this step intuitively adds C to the current proof attempt). Afterwards, the one-step lookahead is updated, by generating all possible inferences in the superposition calculus between C and clauses in *Active*. If at some point the empty clause $\bot$ is selected,[3] saturation finishes, and reports that a proof was found. If at some point no further clause can be selected since *Passive* is

[2] We will later on introduce variants of the given-clause algorithm, called Discount and Otter. For these variants, *Passive* has a slightly different interpretation, as clauses in Passive are potentially derived using additional inferences.

[3] As an optimization, it is sufficient to stop as soon as $\bot$ is generated, without adding it to *Passive* and selecting it.

empty, then saturation finishes, and reports that no proof was found although all clauses have been saturated. Finally, if the time limit is reached, saturation aborts, and reports that no proof was found.

5.1.4 Clause Selection Heuristics

One of the most important choices in the given-clause algorithm is which clause selection heuristics to use. For a comprehensive description and evaluation of existing clause selection heuristics, see [SM16].

To ensure that the given-clause algorithm is complete – in the sense that the empty clause can be derived (given enough time) if the set of input axioms is unsatisfiable and a complete inference system is used – we require the clause selection heuristics to be *fair* [NR01], which intuitively means that no clause stays for an infinite number of rounds in *Passive* without being selected.

Most clause selection heuristics order clauses with respect to one or more features of the clause and/or its derivation. Two such features are the number of symbols in a clause (also known as the *weight* of the clause) and the timepoint at which the clause was derived (also known as the *age*[4] of the clause). The de facto standard clause selection heuristic used in modern saturation-based theorem proving is the *age-weight-based clause selection heuristics* [RV03], which alternates between selecting the clause with the smallest age and selecting the clause with the smallest weight using a fixed ratio (we assume that no value in the ratio is 0). The basic understanding of age-weight selection is that it performs a blend between the best-first and the breadth-first search paradigms. Clauses of small weight are considered better, because they are closer to the ultimate goal – the empty clause of weight zero – than the larger ones. They also tend to produce small clauses as children, serve as stronger simplifiers on average, and are computationally cheaper to process. In contrast, the age feature performs a breadth-first exploration and ensures fairness.

5.1.5 Redundancy

The presented naive version of the given-clause algorithm would be very inefficient as applications of all possible inferences will quickly blow up the search space. It can however be made efficient by exploiting a powerful concept of *redundancy*: deleting so-called *redundant* clauses from the search space while preserving completeness of Sup. A clause C in a set S of clauses (i.e., in the search space) is *redundant* in S, if there exist clauses $C_1, \ldots, C_n$ in S, such that $C \succ C_i$ and $C_1, \ldots, C_n \models C$. That is, a clause C is redundant in S if it is a logical consequence of clauses that are smaller than C w.r.t. $\succ$. It is known that redundant clause can be removed from the search space, without affecting the completeness of superposition-based proof search [BG94, WTRB20]. For

[4] While the term *age* is commonly used in the literature to refer to this feature, *date-of-birth* would be more appropriate. In particular, following the existing literature, clauses, which are earlier generated, confusingly have smaller age.

this reason, state-of-the-art saturation-based theorem provers not only generate new clauses using the inference rules from Sup (we call these rules from now on *generating inference rules*), but also delete redundant clauses during proof search by using so-called *simplifying inference rules* and *deletion inference rules*. We use the term *reduction inference rule* to refer to both simplifying and deletion inference rules.

Simplification Rules. A *simplifying inference* is an inference in which one premise C_i becomes redundant after the addition of the conclusion C to the search space, and hence C_i can be deleted. In what follows, we will denote deleted clauses by drawing a line through them and refer to simplifying inferences as *simplification rules*. The premise C_i that becomes redundant is called the *main premise*, whereas other premises are called *side premises* of the simplification rule. Intuitively, a simplification rule simplifies its main premise to its conclusion by using additional knowledge from its side premises.

Deletion Rules. Even when simplification rules are in use, deleting more/other redundant clauses is still useful to keep the search space small. For this reason, in addition to simplifying and generating rules, theorem provers also use *deletion rules*: a *deletion rule* checks whether clauses in the search space are redundant due to the presence of other clauses in the search space, and removes redundant clauses from the search space.

One example of a simplification rule is *demodulation*, also called *rewriting by unit equalities*. Demodulation is the following inference rule:

$$\frac{l \simeq r \qquad \cancel{L[t] \vee C}}{L[r\sigma] \vee C}$$

where $l\sigma = t$, $l\sigma \succ r\sigma$, and $L[t] \vee C \succ (l \simeq r)\sigma$, for some substitution σ. It is easy to see that demodulation is a simplification rule. Moreover, demodulation is a special case of a superposition inference where the main premise of the inference is deleted. However, unlike a superposition inference, demodulation is not restricted to selected literals.

5.2. Example Consider the clauses $C_1 = f(f(x)) \simeq f(x)$ and $C_2 = P(f(f(c))) \vee Q(d)$. Let σ be the substitution $\sigma = \{x \mapsto c\}$. By the subterm property of $\succ$, we have $f(f(c)) \succ f(c)$. Further, as equality literals are smaller than non-equality literals, we have $P(f(f(c))) \vee Q(d) \succ f(f(c)) \simeq f(c)$. We thus apply demodulation and C_2 is simplified into the clause $C_3 = P(f(c)) \vee Q(d)$:

$$\frac{f(f(x)) \simeq f(x) \qquad \cancel{P(f(f(c))) \vee Q(d)}}{P(f(c)) \vee Q(d)}$$

□

An example of a deletion rule is *subsumption*, which removes subsumed clauses from the search space. Given clauses C and D, we say C *subsumes* D if there is some substitution σ such that $C\sigma$ is a submultiset of D, that is $C\sigma \subseteq_M D$.

5.3. Example Let $C = P(x) \vee Q(f(x))$ and $D = P(f(c)) \vee P(g(c)) \vee Q(f(c)) \vee Q(f(g(c))) \vee R(y)$ be clauses in the search space. Using $\sigma = \{x \mapsto g(c)\}$, it is easy to see that C subsumes D, and hence D is deleted from the search space. □

Saturation with redundancy. There exist different strategies to extend the given-clause algorithm with reduction rules [Vor01]. The two main approaches are variants of the *Otter algorithm* and the *Discount algorithm*. The Otter approach keeps the invariant that all reductions between clauses in *Active* ∪ *Passive* have been performed. This approach spends a lot of time with simplifications (in particular with reducing passive clauses by passive clauses[5]), but allows more simplifications and ensures that clause selection acts on the simplified clauses. Variants of the Discount approach keep the invariant that all reductions between clauses in *Active* have been performed. In contrast to the Otter approach, variants of the Discount approach process clauses much faster, but have the disadvantages that (i) fewer simplifications are performed, and (ii) clause selection might pick suboptimal clauses as clauses in *Passive* are not fully simplified yet.

Forward- and backward-reductions. Let the *reduction set* be *Active* ∪ *Passive* for Otter and *Active* for Discount. Both for Otter and Discount the invariants on the reduction set are maintained as follows: Whenever we insert a new clause C into the reduction set, we first reduce C by clauses $C_1, \ldots, C_k$ which are already in the reduction set. We call such a reduction a *forward* reduction, as the clauses $C_1, \ldots, C_k$ were added to the reduction set before C is added to the reduction set, so the reductions proceed *forward in time*. Secondly, we use C to reduce existing clauses $C_1, \ldots, C_k$ in the reduction set. We call such a reduction a *backward* reduction, as the clause C was added to the reduction set after $C_1, \ldots, C_k$ were added to the reduction set, so the reductions proceed *backward in time*. The distinction between forward and backward reduction rules is important: The main bottleneck for an efficient implementation of reduction inference rules is to retrieve the simplifying clauses in the former case and to retrieve the simplified clauses in the latter case. Both these retrievals are implemented using advanced indexing techniques [SRV01]. Their implementation and efficiency can differ a lot, and in particular backward-reduction rules can sometimes be significantly slower compared to the corresponding forward-reduction rules. As a result, state-of-the-art implementations sometimes realize a given reduction rule by default only as a forward-reduction rule.

5.1.6 Reasoning with Background Theories

The superposition-calculus introduced in the previous subsections is sound and complete (with respect to first-order logic with equality). To reason in first-order logic with equality *modulo some background theory* $\mathcal{T}$, we modify the approach as follows: (i) we extend the set of input clauses with a (best-effort) axiomatization of $\mathcal{T}$, that is, with

[5] The number of clauses in *Passive* is often magnitudes larger than the number of clauses in *Active*.

a set of (clausal) axioms, the so-called *theory axioms*, and (ii) we introduce theory-aware simplification rules[6]. We require that each theory axiom is a tautology in $\mathcal{T}$, and we require that each introduced simplification rule is sound in $\mathcal{T}$. As a consequence, the resulting calculus is sound for first-order logic with equality modulo $\mathcal{T}$. While the resulting calculus is complete (with respect to first-order logic with equality), it is not necessarily complete for $\mathcal{T}$. In particular, we are most interested in the background theory $\mathbb{D} \cup \mathbb{I}$ of difference logic and integer arithmetic, for which it is known that no complete calculus can exist [Göd31].

5.1.7 Validity Checking

We now recall how superposition-based theorem proving is used to reason about validity, and in particular conclude that a given conjecture formula *Conj* is a consequence from a set of axiom formulas $\mathcal{A}$ in first-order logic with equality modulo $\mathcal{T}$.

We know that a conjecture *Conj* follows from a set of axioms $\mathcal{A}$ in $\mathcal{T}$, if and only if the conjunction of the axioms in $\mathcal{A}$ and $\neg Conj$ is unsatisfiable in $\mathcal{T}$. Furthermore, we can use a *CNF-transformation* [PG86, AW13, RSV16] to transform any formula into an equisatisfiable set of clauses. We therefore use the following approach to establish a validity claim: (i) we start with the formula $\mathcal{F}$ consisting of the conjunction of the axioms in $\mathcal{A}$ and $\neg Conj$ (ii) we execute the *preprocessing phase*, where we transform $\mathcal{F}$ into a set of clauses $\mathcal{C}$ using a CNF-transformation, extend $\mathcal{C}$ with an axiomatization of $\mathcal{T}$, and potentially perform other preprocessing steps, and (iii) we perform the *saturation phase*, where we use the given-clause algorithm to show that $\mathcal{C}$ is unsatisfiable. The saturation phase terminates in either of the following three cases: (i) the empty clause $\bot$ is derived (hence, a refutation of $\mathcal{A} \cup \neg\{Conj\}$ was found, which means that *Conj* follows from $\mathcal{A}$ in first-order logic with equality modulo $\mathcal{T}$), (ii) no more clauses are derived and the empty clause $\bot$ was not derived (hence, there exists no refutation of $\mathcal{A} \cup \neg\{Conj\}$, which means that *Conj* does not follow from $\mathcal{A}$ in first-order logic with equality modulo $\mathcal{T}$, if the used inference system together with the used axiomatization is complete for $\mathcal{T}$), or (iii) an a priori given time/memory limit on the Vampire run is reached (hence, it is unknown whether *Conj* follows from $\mathcal{A}$ in first-order logic with equality modulo $\mathcal{T}$).

We can also look at reasoning about validity statements from the perspective of the generated derivations. In the preprocessing phase, a superposition-based prover generates a derivation from $\mathcal{A} \cup \neg\{Conj\}$, such that each sink-node[7] of the DAG is a clause[8]. Then, the prover enters the saturation phase, where it incrementally extends the existing derivation with the given clause algorithm using the sink-nodes from phase (ii) as input clauses.

[6] For such inferences, we modify the notion of redundancy to use logical consequence *modulo* $\mathcal{T}$.
[7] A sink-node is a node such that no edge emerges out of it.
[8] Note that preprocessing inferences do operate on formulas, which are not necessarily clauses.

5.2 Design of Vampire

In this section, we discuss how the design of the state-of-the-art superposition-based theorem prover Vampire [KV13] realizes the abstract design ideas of superposition-based theorem proving. In particular, we emphasize the parts where Vampire offers a choice of algorithmic solutions.

Vampire supports two simplification orderings: a transfinite version [KMV11] of the *Knuth-Bendix Ordering (KBO)* [KB83, Löc06a], and the *Lexicographic Path Ordering (LPO)* [Kam80, BN99, Löc06b]. In both orderings, equality literals are fixed to be always smaller than any other literals.

For literal selection, Vampire uses several heuristics [HRSV16], from which we now highlight the most important ones. The heuristics referred to as `10` in [HRSV16] represents the default literal selection strategy, which is (i) well-behaved[9], (ii) stable against small changes in the search space, and (iii) works well in practice. As an alternative, one can use the *lookahead-selection strategy*, referred to as `11` in [HRSV16]. This heuristics is interesting, as it sometimes outperforms the heuristics `10`. While it is well-behaved, it has the disadvantage that its computation depends on the current state of the search space, which makes it very unstable against small changes to the order in which the search space is explored. Furthermore, there are two not-well-behaved variants of the heuristics `10` resp. `11`, which are referred to as `1010` resp. `1011` in [HRSV16]. While the heuristics `1010` is not well-behaved, it is still quite stable in our experience. The heuristics is interesting, as it is not affected by the (in our context not necessarily optimal) choice in Vampire to force equalities to be always smaller in the literal ordering than other literals. Finally, the heuristics `1011` is another heuristics which works very well for many examples. It has the huge disadvantage, that it is not only not well-behaved but also highly unstable against small changes to the search space, which makes it hard to control while tuning other options.

Vampire uses the superposition calculus Sup, denoted in Figure 5.1. As an optimization, it realizes the superposition inference rule as *Simultaneous superposition* [Sch13, DK20]. Furthermore, Vampire provides an additional generating inference rule *unit resulting resolution* [OMW76]. This rule is redundant from the perspective of completeness, but can still be useful to discover proofs quickly.

The given-clause algorithm is realized in Vampire in three variants: The already discussed approaches Otter and Discount, and an optimization of Otter, called the Limited Resource Strategy (LRS) [RV03], which should be seen as the default variant of the given-clause algorithm used in Vampire. LRS almost always outperforms Otter. While it also often outperforms Discount, there are still many examples, where the roles are switched, with Discount outperforming LRS.

Vampire provides various reduction inference rules and supports to use arbitrary combinations of them. On the one hand, several unary reduction inference rules are provided,

[9] As defined in Subsection 5.1.2.

the most important ones being *tautology deletion*, *duplicate literal removal*, *condensation*, *trivial inequality removal*, *interpreted simplification*, and *distinctness-* and *injectivity-simplification for term algebras*. These rules can be applied very efficiently, therefore Vampire applies them exhaustively as soon as a clause is generated, independently from the chosen variant of the given-clause algorithm. On the other hand, Vampire uses several binary reduction inference rules: *forward- and backward-demodulation* [BG94], *forward- and backward-subsumption*[10] [BG94], and *forward- and backward-subsumption resolution* [BGML01].

To control clause selection, Vampire uses a variant of the standard age-weight based clause selection heuristics, where the age of a clause C is measured as the depth of the derivation of C (simplification inferences are ignored during the depth computation). Furthermore, several experimental features can be used to modify the clause weight used for clause selection. In particular, one can choose to multiply the weight of each clause, which is not derived from the conjecture, by some fixed constant value.

Vampire provides native support for several background theories, in particular for (linear and nonlinear) integer arithmetic and term algebras. For these theories, Vampire (i) adds a set of theory axioms to the input axioms, and (ii) uses custom unary reduction inference rules. The theory axioms used for integer arithmetic and term algebras are described in [RS17] resp. [KRV17]. For integer arithmetic, several other experimental approaches can additionally be used [RS17, RSV18].

Finally, Vampire provides an advanced architecture [Vor14] to interleave saturation with clause splitting [Wei01, RV01].

5.3 Tuning Vampire to Trace Logic with Existing Options

Tuning a superposition-based prover to a given domain can dramatically improve its performance, in particular, if there exists domain-specific knowledge, which can be exploited to guide proof search. In this section, we discuss how to tune Vampire to the trace logic domain, using appropriate choices for the techniques presented in Section 5.2. While some of these choices are currently only available in Vampire, we argue that most of our choices should also apply to other state-of-the-art saturation-based theorem provers.

We use the (transfinite) KBO ordering, as it works well together with other parts of the prover, in particular with (age-) weight-based clause selection.

To handle literal selection, we employ the heuristics `10` and `1010`. The heuristics `10` is appealing, as it is well-behaved and stable, and works well on the trace logic domain. Surprisingly, we encountered many examples in the trace logic domain, which were only provable using the heuristics `1010`. We suspect that for these examples, the heuristics `10`

[10] Technically, subsumption is not a reduction inference rule, as it is not covered by the standard redundancy criterion from [BG94]. It is possible to adapt the redundancy criterion so that it covers subsumption too [WTRB20].

encountered the problem that an important clause was not selected by clause selection, as its weight was too large, which was caused by several equality literals, which piled up in the clause during the derivation, as the literal ordering makes all equality literals smaller than any non-equality literals. We suspect that in contrast, the heuristics `1010` could resolve away these equality literals earlier, and therefore kept the relevant clauses smaller, which in turn led to an earlier selection of these clauses by the clause selection heuristics. We also experimented with the heuristics `11` and `1011`. While they worked reasonably well, in our experience they did not fundamentally improve proof search. As their instability against small changes to the search space would have made it very difficult to optimize the choices for other options, we did not explore them further.

As our application domain is "nearly-Horn", in the sense that only a few case distinctions are required, we use the redundant generating inference rule unit resulting resolution. In our experience, unit resulting resolution is very beneficial in the trace logic domain.

We use the LRS algorithm, which in our experience consistently outperforms Otter. We also tried to use Discount, but could not improve on LRS with Discount on the trace logic domain, even though we optimized Discount by varying several other options. We conjecture that one reason for the better performance of LRS compared to Discount is that LRS can backward simplify clauses in *Passive* by clauses added to *Active*. Such simplifications occur frequently in our domain. As a result, weight-based clause selection is more accurate in LRS than in Discount, since it is computed on exhaustively simplified clauses.

To fight the explosion of the search space, we use all reduction inference rules discussed in Section 5.2. While in general there exist benchmarks, where the evaluation of some of these inferences rules takes an excessive amount of time, we did not observe this problem for the trace logic domain.

We use the age-weight-based clause selection heuristics and alternate between selecting the clause with the smallest age and selecting the clause with the smallest weight using a ratio of 1:1. Furthermore, we focus proof search towards the conjecture by doubling the weight used for clause-selection for all clauses, which do not derive from the conjecture. That way, clauses, which derive from the conjecture, will be preferred whenever the clause selection heuristics selects by weight.

Interestingly, in our experiments on the trace logic domain, it did not help to use the Avatar architecture, even though we spent a lot of time optimizing strategies so that Avatar could shine. This result might seem surprising. We argue that it is reasonable, as (i) the trace logic domain contains a lot of quantification, which limits the amount of splitting possible by Avatar, and (ii) we carefully introduce definitions in the trace logic encodings (which amounts to a manual upfront splitting of clauses) and therefore already handle the most important splittings without depending on Avatar. Not having to use Avatar is highly beneficial from a user perspective, as (i) proof search is much more stable and predictable if Avatar is not used, (ii) we can manually inspect the proofs found by Vampire *efficiently*, in contrast to the proofs produced by Avatar, which can

$$
\begin{array}{ll}
\forall it^{\mathbb{D}}. & (it \simeq 0 \vee \mathtt{succ}(\mathtt{pred}(it)) \simeq it) \\
\forall it^{\mathbb{D}}. & 0 \not\simeq \mathtt{succ}(it) \\
\forall it_1^{\mathbb{D}}, it_2^{\mathbb{D}}. & (it_1 \simeq it_2 \rightarrow \mathtt{succ}(it_1) \simeq \mathtt{succ}(it_2)) \\
\forall it^{\mathbb{D}}. & 0 < \mathtt{succ}(it) \\
\forall it_1^{\mathbb{D}}, it_2^{\mathbb{D}}. & (it_1 < \mathtt{succ}(it_2) \leftrightarrow (it_1 \simeq it_2 \vee it_1 < it_2)) \\
\forall it_1^{\mathbb{D}}, it_2^{\mathbb{D}}. & (it_1 < it_2 \leftrightarrow \mathtt{succ}(it_1) < \mathtt{succ}(it_2)) \\
\forall it_1^{\mathbb{D}}, it_2^{\mathbb{D}}, it_3^{\mathbb{D}}. & ((it_1 < it_2 \wedge it_2 < it_3) \rightarrow it_1 < it_3) \\
\forall it_1^{\mathbb{D}}, it_2^{\mathbb{D}}, it_3^{\mathbb{D}}. & ((it_1 < \mathtt{succ}(it_2) \wedge it_2 < it_3) \rightarrow it_1 < it_3) \\
\forall it_1^{\mathbb{D}}, it_2^{\mathbb{D}}, it_3^{\mathbb{D}}. & ((it_1 < it_2 \wedge it_2 < \mathtt{succ}(it_3)) \rightarrow it_1 < it_3) \\
\forall it_1^{\mathbb{D}}, it_2^{\mathbb{D}}, it_3^{\mathbb{D}}. & ((it_1 < \mathtt{succ}(it_2) \wedge it_2 < \mathtt{succ}(it_3)) \rightarrow it_1 < \mathtt{succ}(it_3)) \\
\forall it_1^{\mathbb{D}}, it_2^{\mathbb{D}}. & (it_1 < it_2 \vee it_1 \simeq it_2 \vee it_2 < it_1) \\
\forall it^{\mathbb{D}}. & it \not< it
\end{array}
$$

Figure 5.2: List of theory axioms used to specify difference logic over natural numbers.

not be inspected in a reasonable amount of time, and (iii) we can visualize saturation attempts (cf. Chapter 8), which allows us to get useful insights for further tuning even from failed proof attempts.

To enable integer reasoning, we use the theory axioms and reduction inference rules for integer arithmetic provided by Vampire. To support the background theory of difference logic, we use the term algebra $(\mathbb{N}, 0, \mathtt{succ})$ of natural numbers, extend it with a symbol $<$ to denote the standard order relation on natural numbers, and extend the theory axioms, which are already internally added by Vampire for this term algebra, with additional axioms for the order relation $<$. A list of the additionally added axioms is presented in Figure 5.2.

CHAPTER 6

Layered Clause Selection for Saturation-based Theorem Proving

In Chapter 5, we discussed how to apply the superposition-based theorem prover Vampire to reason in the trace logic domain, by tuning it using existing options (Section 5.3). In initial experiments with this tuned setup, we observed that Vampire would still fail to prove most of the examples of the trace logic domain. While manually inspecting some of the failed proof attempts, we realized that Vampire's high-level exploration of the proof space was ineffective. In particular, Vampire would heavily focus on searching for proofs containing (i) almost only theory reasoning, and/or (ii) an excessive amount of case-distinctions. In contrast to the proofs Vampire was searching for, most proofs in the trace logic domain *only require light-weight theory reasoning and light-weight reasoning with case distinctions*. We conjecture that the key ingredient to efficient reasoning in the trace logic domain is to guide the prover using this domain-specific knowledge.

In this chapter, we will realize this idea, as follows. First, we will introduce a framework for *layered clause selection*, which uses *split heuristics* to adapt existing clause selection heuristics with arbitrary clause features, by ensuring that for a fixed percentage of clause selections, clauses are selected which have small values for the given clause features. Secondly, we will identify clause features, which heuristically approximate (i) the amount of theory reasoning in the derivation of a clause (Section 6.2), and (ii) the number of case distinctions in the derivation of a clause (Section 6.3), and instantiate the layered clause selection with these features.

Afterwards, we will introduce an additional feature, which heuristically approximates the relevance to the conjecture of a clause (Section 6.4), to focus proof search towards the conjecture. Additionally, we will describe a feature to improve proof search in the setting

of Avatar (Section 6.5). While the resulting clause selection heuristics does not perform up to our initial expectations, we still include it for completeness of the presentation. We conclude the chapter by evaluating the described heuristics on standard benchmark sets (Section 6.6).

While the techniques described in this chapter are motivated by the trace logic domain, they should also be beneficial in other domains that have characteristics similar to the trace logic domain. In particular, we conjecture that the techniques of this chapter should be highly effective in many verification domains, e.g. on the examples from [BBC+19].

6.1 Layered Clause Selection using Split Heuristics

6.1.1 The Age-Weight-Based Heuristic

We start with a formal definition of the age-weight clause selection heuristics, which was already introduced in Subsection 5.1.4.

6.1. Definition (Age-weight clause selection heuristic) For any clause C, define the age $age(C)$ as the depth of the derivation tree of C[1] and define the weight $weight(C)$ as the number of symbols[2] of C. Let further $r_a : r_w$ be a list of two positive integer values. Then the *age-weight clause selection heuristic* $aw(r_a : r_w)$ alternates between selecting a clause C with the smallest age $age(C)$ and selecting a clause C with the smallest weight $weight(C)$ using the ratio $r_a : r_w$.

6.1.2 Split Heuristics

6.2. Definition (Split heuristics) Let μ be a real-valued *clause evaluation feature* such that preferable clauses have a low value of $\mu(C)$, and let the *cutoffs* $c_1, \ldots, c_k$ be monotonically increasing real numbers with $c_k = \infty$. Furthermore, let the *ratio* $r_1 : \ldots : r_k$ be a list of positive integer values, and let finally cs be an arbitrary clause selection heuristic.

A *split heuristic* groups clauses into sets $G_1, \ldots, G_k$, and selects clauses by alternating selection from $G_1, \ldots, G_k$ using the ratio $r_1 : \ldots : r_k$. The selection from each such set G_i is performed using cs. We define two *modes* of split clause selection heuristic, which differ in how they group clauses:

- The *monotone split heuristic* *mono-split*$(\mu, c_1, \ldots, c_k, r_1 : \cdots : r_k, cs)$ uses sets $G_i := \{C \mid \mu(C) \leq c_i\}$ for $i = 1, \ldots, k$.
- The *disjoint split heuristic* *disj-split*$(\mu, c_1, \ldots, c_k, r_1 : \cdots : r_k, cs)$ uses sets $G_1 := \{C \mid \mu(C) \leq c_1\}$, and $G_i := \{C \mid c_{i-1} < \mu(C) \leq c_i\}$ for $2 \leq i \leq k$.

[1] This corresponds to how age is defined in Vampire. More precisely, one uses only the depth with respect to generating inferences. Reductions do not alter the age of a reduced clause.

[2] Including multiplicities. As a variation, different kinds of symbols (such as the variables, the predicate symbols, or the constants) may weigh more than others [SM16].

6.3. Example Consider the clause selection heuristic $\mathit{mono\text{-}split}(\mu, 0, 1, \infty, 3:1:1, \mathit{aw}(1:1))$. This heuristic will select 3 out of 5 times a clause C such that $\mu(C) \leq 0$, 1 out of 5 times a clause C such that $\mu(C) \leq 1$, and 1 out of 5 times an arbitrary clause. On "layer one", e.g., 3 out of 10 times the clause C with the smallest age among the clauses C with $\mu(C) \leq 0$ is selected, or 1 out of 10 times the clause with the smallest weight out of all clauses is selected. □

Split heuristics allow to adapt an existing clause selection heuristic *cs* to take into account the clause feature μ. We now discuss how to pick the mode, the cutoffs, and the ratios. We observed two kinds of clause features:

First, there are features where clauses with low feature value are more likely to contribute to the proof search (this is the case for the features dist_{th}, dist_{Horn}, and dist_{SInE}, discussed in Section 6.2, Section 6.3, resp. Section 6.4). For features of this kind, one can use a monotone split heuristic. A good starting point for cutoffs and ratios is c_1, ∞ and $1:1$, resp., where c_1 is the feature value we expect to obtain for the empty clause $\bot$ (based on domain knowledge and experience). One can extend the cutoffs by introducing one or two additional cutoffs close to c_1, in order to smooth the transition between c_1 and ∞, and extend the ratio accordingly. It can also make sense to vary the ratio, although, in our experience, it is more important to identify good cutoffs than to fine-tune the ratio.

Secondly, there are features where clauses with low feature value are not necessarily more likely to contribute to the proof search, but are less likely to have low weight (this is the case for the feature dist_{AV} discussed in Section 6.5). As a consequence, it does not make sense to compare clauses, which have different feature values, by weight. In such a case, one can use a split heuristic in disjoint mode. Varying the cutoffs and ratios of disjoint split heuristics has a less predictable effect than for the monotone split heuristics and needs to be fine-tuned on a case-by-case basis.

6.1.3 Nesting Split Heuristics

As split heuristics are parameterized by an arbitrary clause selection heuristic, we can build clause selection heuristics containing nestings of split heuristics. Such clause selection heuristics are powerful, as they allow us to easily combine different features.

6.4. Example Consider the nested split heuristic

$$\begin{gathered}\mathit{mono\text{-}split}(\mu_1, 0, 1, \infty, 15:4:1,\\ \mathit{mono\text{-}split}(\mu_2, 1, \infty, 5:1,\\ \mathit{aw}(1:1))).\end{gathered}$$

The resulting clause groupings and frequencies to pick from these groups are visualized in Figure 6.1. The nested split heuristics form a tree. Each leaf node of the tree represents a set of clauses, from which clauses are selected using $\mathit{aw}(1:1)$. We can see that the

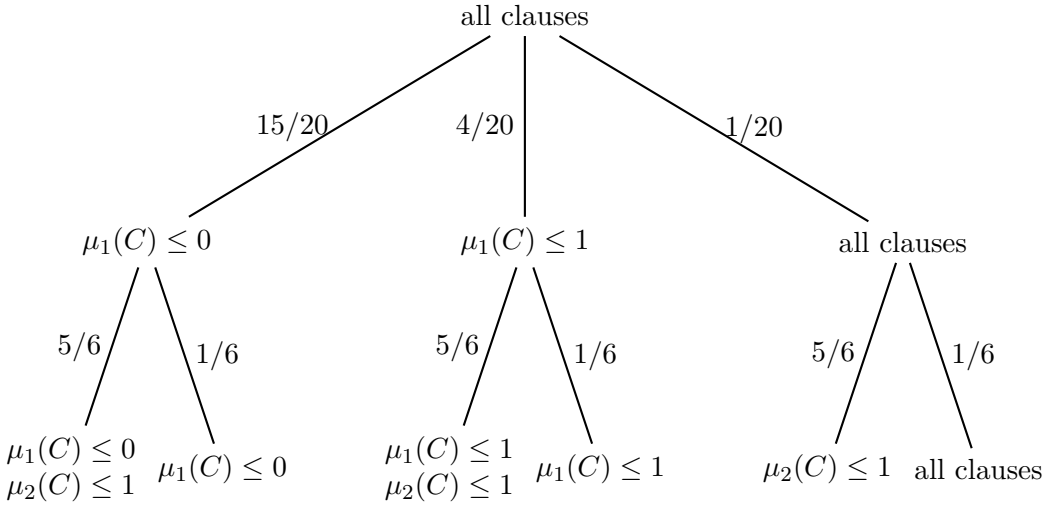

Figure 6.1: Demonstrating nested split heuristics.

leftmost leaf node of the tree represents all clauses C with $\mu_1(C) \leq 0$ and $\mu_2(C) \leq 1$. We pick clauses from this leaf using $aw(1:1)$ in $15/20 * 5/6 = 5/8$ of the cases. □

Note that each split heuristic h provides a horizontal dimension consisting of the groups of h. The nesting of different split heuristics itself provides a vertical dimension.

6.1.4 Implementation

In this subsection, we briefly discuss how to implement clause selection heuristics. As typical runs of saturation algorithms can include several million clause selections, we strive to implement these heuristics efficiently.

An aw-heuristic with ratio $r_{age} : r_{weight}$ can be implemented as a container AW as follows. The container AW internally keeps two priority queues[3] Q_a, Q_w, where both Q_a and Q_w store all the clauses of AW, Q_a keeps its clauses ordered by *age* and Q_w keeps its clauses ordered by *weight*. The container AW determines whether it should select the next clause from Q_{age} or Q_{weight} by using a weighted round-robin scheme with ratio $r_a : r_w$, and the selection from the chosen queue proceeds by popping the first element (i.e. a clause) from that queue and deleting the corresponding record (of that clause) from the other queue.

The heuristic *mono-split*$(\mu, c_1, \ldots, c_k, r_1 : \cdots : r_k, cs)$, resp. *disj-split*$(\mu, c_1, \ldots, c_k, r_1 : \cdots : r_k, cs)$, can be implemented as a container SH as follows. Assume that cs is implemented using a container CS. The container SH keeps k instances $CS_1, \ldots, CS_k$ of CS, where container CS_i contains all clauses of group G_i of SH. The container SH determines from which of the sub-containers $CS_1, \ldots, CS_k$ it should select the next

[3] In Vampire, these priority queues are implemented as skip lists.

clause using a weighted round-robin scheme with ratio $r_1 : \cdots : r_k$, and then delegates clause selection to that CS_i.

6.1.5 Discussion

We believe that the nesting of split heuristics is a great conceptual tool for composing independent ideas on how to improve clause selection into a single compound heuristic. This can be already seen with two layers, where the time-tested age-weight selection serves as a building block for more powerful/refined heuristics, and can get further pronounced with additional nestings, as demonstrated by our experiments (see Section 6.6).

Nevertheless, it is not hard to see that computationally, the layered scheme can be essentially "compiled down" to multiple level-one queues. More precisely, one needs an extension of the typical level-one queue arrangement, such as the one implemented in E [Sch13], to allow clause queue *content filtering* by clause properties. This means that one needs to be able to set up a clause queue that only contains those clauses that satisfy a given property P. Such property P could be, e.g., $P(C) = \mu_1(C) \leq 0 \wedge \mu_2(C) \leq 1$ to define the age and weight level-one queues corresponding to the left-most leaf in Figure 6.1. The reason why clause queue content filtering has until now not been used (to the best of our knowledge) in saturation-based provers is probably that in the standard perspective each clause queue is meant to provide an independent view of the whole set of passive clauses and not just a subset thereof (which would complicate reasoning about completeness if left further unconstrained).[4]

6.2 Feature: Amount of Theory Reasoning

Many domains, including the trace logic domain and the examples from [BBC+19], require a prover to perform quantified reasoning in a given (background) theory. The standard solution to provide a prover with support for reasoning in such a theory is to extend the input axioms of the problem with an explicit axiomatization of the corresponding theory. There are two related problems caused by this approach: First, the theory axioms generate a huge number of consequences, as the theory axioms are repeatedly combined either with themselves or with other axioms, and therefore blow up the search space. Secondly, many of these generated consequences have a small weight. If a standard age-weight-based heuristic is used for clause selection, those consequences are therefore often selected, as selection by weight will favor them. While manually inspecting proofs for problems of the trace logic domain and for problems from [BBC+19], we observed that the amount of theory reasoning actually required to prove these problems is small. As a result, the prover spends most of its proof search in a part of the search space, where the chances to find a clause relevant for the proof are low. We are therefore facing the challenge of guiding the proof search, so that the prover does not spend too

[4] We note that clause *priority functions* of E [Sch02] allow the user to order clauses on a particular queue such that those clauses satisfying a given property P are all considered smaller than those that satisfy $\neg P$.

much time with theory reasoning, but at the same time still finds proofs containing a small amount of theory reasoning.

In the remainder of this section, we present a solution for this challenge. In a nutshell, our solution consists of a clause feature $dist_{th}$, which measures the amount of theory reasoning in the derivation of a clause, and a corresponding clause selection heuristic based on split heuristic and $dist_{th}$. We assume that the input problem is given as a set of axioms, where the axioms corresponding to the axiomatization of the theory are distinguished.

We start by formalizing the amount of theory reasoning in the derivation of a clause C as the ratio of the number of theory axioms and the number of all axioms in the derivation-DAG of C. Computing these numbers exactly for the derivation of each clause is potentially expensive, since it requires for each clause a traversal of the derivation-DAG of the clause. We instead approximate those numbers by treating the derivation-DAG as a tree, for which we can compute the numbers using running sums, as follows.

6.5. Definition For a theory axiom C, define both $thAx(C)$ and $allAx(C)$ as 1. For a non-theory axiom C, define $thAx(C)$ as 0 and $allAx(C)$ as 1. For a derived clause C with parent clauses $C_1, \ldots, C_n$, define $thAx(C)$ as $\sum_i thAx(C_i)$ and $allAx(C)$ as $\sum_i allAx(C_i)$. Finally, we set $frac(C) := thAx(C)/allAx(C)$.

With these notations at hand, we identify proofs that only need a small amount of theory reasoning with the proofs where $frac(\bot)$ is at most $1/d$, for some small positive integer value d ($\bot$ here denotes the empty clause).

Next, we present a clause feature $dist^d_{th}$ which approximates the likeliness that a given clause C occurs in a proof where $frac(\bot)$ is at most $1/d$. The clause selection feature $dist^d_{th}$ is parameterized by the value d and measures the number of non-theory axioms which the derivation of C would need to contain additionally to achieve a ratio of at most $1 : d$.

6.6. Definition Let d be a positive integer value. Then $dist^d_{th} : Clauses \to \mathbb{N}$ is defined as

$$dist^d_{th}(C) := max(thAx(C) \cdot d - allAx(C), 0).$$

The feature $dist^d_{th}$ satisfies several properties, which we think are favorable: (i) if the derivation of a clause C consists only of several axioms, then $dist^d_{th}(C)$ is small, (ii) if derivations of clauses C_1, C_2 are combined into a derivation of clause C, and if both $dist^d_{th}(C_1) > 0$ and $dist^d_{th}(C_2) > 0$, then $dist^d_{th}(C) > dist^d_{th}(C_1)$ and $dist^d_{th}(C) > dist^d_{th}(C_2)$, and (iii) if derivations of clauses C_1, C_2 are combined into a derivation of clause C, and if $frac(C_1) = 1/d$, then $dist^d_{th}(C) = dist^d_{th}(C_2)$. Note that $frac$ itself does not fulfill these properties.

6.7. Example Consider a clause C_1, such that $thAx(C_1) = 3$ and $allAx(C_1) = 5$. Consider further a clause C_2, such that $thAx(C_2) = 100$ and $allAx(C_2) = 200$. In-

tuitively, C_1 is much more likely to occur in a proof with $frac(\bot) = 1/4$. We have $dist^4_{th}(C_1) = 7 < 200 = dist^4_{th}(C_2)$, but $frac(C_1) = 0.6 > 0.5 = frac(C_2)$. □

Finally, we construct a clause selection heuristic, which addresses the challenges presented at the beginning of this section, using the split heuristic from Section 6.1.2 as

$$mono\text{-}split(dist^d_{th}, c_1, \ldots, c_{k-1}, \infty, r_1 : \cdots : r_k, \mu),$$

where d is the positive integer such that $1/d$ is the expected fraction of the proof which we want to find, μ is some clause selection strategy, $c_1, \ldots, c_{k-1}, \infty$ are cutoff values, and $r_1 : \cdots : r_k$ is a ratio. In our experience, varying d has a bigger effect than varying cutoffs and ratios, and setting d to 8 is a reasonable starting point for fine-tuning d. In our experiments, other useful values of d were between 4 and 50.

6.3 Feature: Positive Literals

Saturation-based theorem provers are known to work well on benchmarks where each axiom is a Horn clause, that is, a clause with at most one positive literal. We would like to extend the efficiency of these provers to problems which are nearly Horn, in the sense that there exists a proof of the conjecture of the problem, where the number of positive literals for each clause is small. We can formalize this as follows: The *Horn-distance* $dist_{Horn}(C)$ is defined as

$$dist_{Horn}(C) := max(posLits(C) - 1, 0),$$

where $posLits(C)$ denotes the number of positive literals of C. For a given proof P define

$$dist_{Horn}(P) := \sum_{C \text{ clause in } P} dist_{Horn}(C).$$

We claim that for many application domains, including the trace logic domain and the examples from [BBC+19], most examples are provable using a proof with a small Horn-distance. But if we run a saturation-based theorem prover on such a problem, it can still generate a lot of consequences that contain several positive literals. Such consequences would typically be classified as highly unlikely to contribute to the refutation by human inspection.

We can guide clause selection towards finding proofs with small Horn-distance by instantiating the split heuristic from Section 6.1 as

$$disj\text{-}split(dist_{Horn}, c_1, \ldots, c_{k-1}, \infty, r_1 : \cdots : r_k, \mu),$$

for a given clause selection function μ, cutoffs $c_1, \ldots, c_{k-1}, \infty$ and ratio $r_1 : \cdots : r_k$.

6.4 Feature: SInE-Levels

Several techniques in saturation-based theorem proving [WRC65, RS17, Sch13, SM16] are based on the intuitive idea, that clauses, which are closer related to the conjecture, have a higher probability to participate in a proof of the conjecture. The Sumo Inference Engine (SInE) [HV11] is a well-established algorithm for *selecting premises* for first-order theorem proving, i.e. for the task of reducing—before the start of the search—the possibly large set of input axioms to a more manageable subset of those ones estimated to be most promising for proving a given conjecture. SInE is an iterative algorithm that takes the conjecture[5] and iteratively adds axioms that appear to be most related to the conjecture or to previously added axioms by a similarity metric based on sharing symbols. We define, for every input axiom A, a *heuristical distance* $dist_{SInE}(A)$ *from the goal* G as the iteration number i at which A would be added by SInE to the included axioms for proving G. By definition, $dist_{SInE}(G) = 0$ for the conjecture itself and typically ranges between 1 up to approximately 10 for the non-conjecture input axioms [Sud20]. We will informally refer to the value $dist_{SInE}(F)$ for a particular formula F as its SInE-level.

So far, we defined SInE-levels only for the input axioms and the conjecture. To use them as a feature of arbitrary clauses in proof search, we further define $dist_{SInE}(C)$ of a derived clause as the *minimum* of $dist_{SInE}(P_i)$ over the parents P_i of C. While the choice of the minimum operation may appear arbitrary, note that it has the nice property that $dist_{SInE}(C) = 0$ if and only if C has the conjecture among its ancestors. This is an important "flag" of a clause, typically tracked by a theorem prover for use in various conjecture-directed heuristics, and SInE-levels therefore naturally generalize this flag.

Finally, we derive a clause selection heuristic using the split heuristic from Section 6.1 as

$$\mathit{mono\text{-}split}(dist_{SInE}, c_1, \ldots, c_{k-1}, \infty, r_1 : \cdots : r_k, \mu),$$

for a given clause selection function μ, cutoffs $c_1, \ldots, c_{k-1}, \infty$ and ratio $r_1 : \cdots : r_k$. For instance, we can use cutoffs $0, \infty$ and ratio $1 : r_2$ to ensure that from $r_2 + 1$ clauses at least one clause is selected which has the conjecture among its ancestors.

6.5 Feature: AVATAR-Splits

Avatar [Vor14, RSV15, RBSV16] is a theorem prover architecture in which a saturation algorithm is augmented with a SAT (or an SMT) solver to facilitate an efficient version of clause splitting [Wei01, RV01]. In a nutshell, a first-order clause C is called *splittable* if it can be written as $C = C_1 \vee \ldots \vee C_k$, $k > 1$, such that the individual *components* C_i are pairwise variable-disjoint. The main idea behind splitting is that one can reason about the individual components separately, since for every set of clauses N and every such splittable clause C, $N \cup \{C\}$ is unsatisfiable if and only if $N \cup \{C_i\}$ is unsatisfiable for every $i = 1, \ldots, k$. This is advantageous, as the individual components C_i are smaller than the original clause C and thus promise a strictly faster search.

[5] Also called the *goal* in [HV11] and [Sud20].

While the exact details of how Avatar works are out of the scope of this chapter, the key aspect important here is easy to explain. First-order clauses in Avatar need to keep track of the dependencies on splits from which they were derived. This is done by assigning to each clause C a set of dependencies D_C, denoted $C \leftarrow D_C$, where a dependency $d \in D_C$ is some identifier of a performed split registered elsewhere in the architecture. Clauses from the input have their dependency set initialized as empty and the dependency set of a derived clause is computed as the union of the dependencies of its parents. Then, when a clause such as $C_1 \vee C_2 \leftarrow D_C$ is split, with C_1 and C_2 variable-disjoint, the prover may continue reasoning with $C_1 \leftarrow D_C \cup \{[C_1]\}$ where $[C_1]$ is the identifier of the dependency on the performed split. Intuitively, each dependency $d \in D_C$ is a choice point for which the prover might need to consider alternatives in the future. This means that a clause with many dependencies corresponds to a logically weaker fact than a clause with fewer ones.

Because the basic setup of Avatar is oblivious to the size of the dependency set of a clause, there is a danger of a strong preference for clauses of small weight (which arise easily with splitting) that nevertheless depend on many splits and are therefore not the best for closing the overall search fast. To potentially mitigate this effect, we propose here to use the size of the dependency set of a clause, $dist_{AV}(C \leftarrow D_C) = |D_C|$, as a feature for split heuristics.

We then construct a clause selection heuristic using the split heuristic from Section 6.1 as

$$\mathit{disj\text{-}split}(\mathit{dist}_{AV}, c_1, \ldots, c_{k-1}, \infty, r_1 : \cdots : r_k, \mu),$$

for a given clause selection function μ, cutoffs $c_1, \ldots, c_{k-1}, \infty$ and ratio $r_1 : \cdots : r_k$.

6.6 Experiments

The techniques developed in this chapter have initially been motivated by the trace logic domain. To understand the effect layered clause selection has for reasoning in the trace logic domain, we will experimentally evaluate layered clause selection on the trace logic domain later as part of a broader experimental evaluation in Chapter 9.

In this section, we will evaluate layered clause selection on the two general domains of the standard benchmark libraries TPTP [Sut17] and SMT-LIB [BFT16]. As the introduced clause selection heuristics encode domain-specific knowledge, we expect them to work well on domains like the trace logic domain or the examples from [BBC+19], where the required proofs share similar characteristics regarding the clause selection features introduced in this chapter. Maybe surprisingly, we will show that even on the general domains of TPTP and SMT-LIB, which by design cover many different reasoning problems of varying characteristics, layered clause selection improves the efficiency of Vampire.

Implementation. We implemented the heuristics described in Sections 6.2–6.5 in the state-of-the-art theorem prover Vampire [KV13]. Our implementation consists of about 1000 lines of C++ code and is part of Vampire[6] starting from version 4.5.

We added the following options to control layered clause selection in Vampire. The options `-thsq`, `-plsq`, `-slsq`, and `-avsq` control whether a split heuristic with feature $dist_{th}$, $dist_{Horn}$, $dist_{SInE}$ resp. $dist_{AV}$ is used (with possible values `on` and `off` and default value `off`). For each of these heuristics, we furthermore added options to control the cutoff (options `-thsqc`, `-plsqc`, `-slsqc`, resp. `-avsqc`), the ratio (options `-thsqr`, `-plsqr`, `-slsqr`, resp. `-avsqr`), and the mode of splitting (options `-thsql`, `-plsql`, `-slsql`, resp. `-avsql`, with values `on` and `off` encoding the usage of the monotone resp. disjoint split mode). The default values for these more fine-grained options are described in Table 6.1.

Benchmarks. We evaluated the extended implementation of Vampire on two sets of problems coming from the TPTP library [Sut17] and from SMT-LIB [BFT16], respectively. In detail, we selected all the first-order problems of the form CNF, FOF, and TF0 (including those with arithmetic) from TPTP version 7.3.0. This gave us 18 294 problems. Additionally, we picked a subset of a recent version (release 2019-05-06) of SMT-LIB consisting of all the problems from the sub-logics that contain quantification and theories, such as ALIA, LRA, NRA, UFDT, ..., except for those requiring bit-vector (BV) or floating-point (FP) reasoning, currently not supported by Vampire. For this SMT-LIB benchmark we obtained 68 234 problems.

Experimental setup. Our experiments were run on our local server with two Intel Xeon Gold 6140 Processors (i.e., with 72 processor threads) and 188GB RAM. We were running 30 instances of Vampire in parallel with no other significant load on the server. To obtain a baseline strategy, denoted as `base`, we modified the default Vampire strategy (which uses Avatar) to use the Discount saturation loop (for stability of results[7]) and the clause selection heuristic $aw(1:10)$ (which in our experience leads in Vampire to good performance with Discount). All other tested strategies extend `base` by applying one or more split heuristics for clause selection on top this setup. With the exception of Experiment 4 we used a time limit of 10 s per problem.[8]

6.6.1 Experiment 1: Testing the Initial Defaults

Searching for good values of the cutoffs, the ratio, and other parameters of split heuristics is rewarding, but requires some experience and a certain amount of experimental "tuning", in particular for general domains. We picked certain default values for the

[6] https://github.com/vprover/vampire

[7] The default Limited Resource Strategy [RV03] is sensitive to timing measurements and repeated runs on the same benchmark under essentially the same conditions may vary a lot.

[8] A list of the selected problems along with other information needed to reproduce our experiments can be found at https://github.com/quickbeam123/LCS4SbTP-materials.

Table 6.1: Default parameters for the heuristics presented in Experiment 1.

tag	feature	d-value	cutoffs	ratio	mode of split
`th`	$dist^d_{th}$	8	$(0, 32, 80, \infty)$	20:10:10:1	monotone
`av`	$dist_{AV}$	–	$(0, \infty)$	1:1	disjoint
`sl`	$dist_{SInE}$	–	$(0, 1, \infty)$	1:2:3	monotone
`pl`	$dist_{Horn}$	–	$(0, \infty)$	1:4	disjoint

Table 6.2: Results of Experiment 1: on the TPTP benchmark (left) and the SMT-LIB one (right).

strategy	solved	Δbase	uniques
`base`	9108	0	9
`th`	9204	96	22
`av`	9160	52	89
`sl`	9525	417	109
`pl`	9289	181	42
`th+av+sl+pl`	9526	418	147
union	10 297	1189	

strategy	solved	Δbase	uniques
`base`	39 943	0	45
`th`	41 841	1898	321
`av`	39 906	−37	116
`sl`	—	—	—
`pl`	40 442	499	95
`th+av+pl`	40 952	1009	287
union	43 169	3226	

parameters of the heuristics introduced in the previous chapters and used these defaults (presented in Table 6.1) in the first experiment, the purpose of which is to demonstrate the basic improvements we obtain from using the presented heuristics and their combinations.

Table 6.1 assigns a *tag*, typeset in the `typewriter` font, to each of our four heuristics when understood as Vampire options used for defining a strategy. Thanks to the possibility of nesting split heuristics, these options can be turned on and off independently from one another. Although a particular fixed order is employed in Vampire to build up the nestings (namely the order, from inside out: `th`, `av`, `sl`, `pl`), we use the operator + to denote possible combinations to suggest that this order is actually irrelevant for the proof search (cf. Section 6.1.5).

The results of the first experiment are shown in Table 6.2, separately for TPTP and for SMT-LIB. We observe that in the case of TPTP all the four new heuristics lead to an improvement in the number of solved problems. This is easiest to see from the always positive column Δbase, which shows the difference of the number of problems solved between the current strategy and `base`. The same success is not fully repeated on SMT-LIB where `th` shows a great improvement, but `av` performs worse than `base`. (Note that we did not run `sl` on SMT-LIB, since the format used in the library does not support specifying the goal.[9])

It should be pointed out that even a strategy that does not improve over `base` in terms

[9]There is, however, interesting work on "guessing the goal" for SMT-LIB problems [RR18], which one could experiment with in the future.

Table 6.3: Combined strategies run on TPTP: number of solved problems and average time spent (on the commonly unsolved problems) maintaining the passive clause container.

strategy	solved	Δbase	#queues	avg. time on unsolved
base	9108	0	2	0.32 s
pl	9289	181	2(*2)	0.32 s
av+pl	9179	71	2(*2)(*2)	0.32 s
av	9160	52	2(*2)	0.33 s
sl	9525	417	2*3	0.58 s
sl+pl	9594	486	2*3(*2)	0.58 s
sl+av+pl	9511	403	2*3(*2)(*2)	0.58 s
sl+av	9560	452	2*3(*2)	0.59 s
th+av+pl	9181	73	2*4(*2)(*2)	1.04 s
th+pl	9338	230	2*4(*2)	1.06 s
th	9204	96	2*4	1.06 s
th+av	9234	126	2*4(*2)	1.08 s
th+av+sl+pl	9526	418	2*4(*2)*3(*2)	1.75 s
th+sl+pl	9601	493	2*4*3(*2)	1.76 s
th+av+sl	9584	476	2*4(*2)*3	1.79 s
th+sl	9557	449	2*4*3	1.80 s

of the number of solved problems could still be valuable for the potential participation in strategy schedules, because of the problems it solves uniquely (as reported in the last column in the tables). For another view of this effect, the last line in the two tables shows the number of problems solved by at least one of the listed strategies, again also compared against base. We can see that the use of split heuristics allows us to solve almost 1200 (more than 3200) problems not solved by base on the two benchmarks, respectively.

6.6.2 Experiment 2: Nesting of the Heuristics

When looking in Table 6.2 at the performance of the combination of the four heuristics (strategy th+av+sl+pl) one can ask why it does not get better at achieving a combined benefit of its constituents. In Experiment 2, we look at this trend closer and especially try to estimate how much time is typically spent on computing the clause selection heuristic and how this depends on the number of heuristics combined.

The report in Table 6.3 is based on TPTP runs of all the 16 possible strategies which combine between zero to four of the heuristics introduced in this chapter. The middle part of the table reports on their performance in terms of the number of solved problems and is comparable to (in fact, a super-set of) the results in Table 6.2 (left). One can notice here that combinations indeed sometimes do not outcompete their constituents.

E.g., `*+av+pl` is always worse than just `*+pl`, which could indicate some unfavorable interactions of the two heuristics (we leave a more detailed study of this phenomenon for future work).

Our main focus in this experiment, however, is on the right part of the table. There, we took the runs on those problems which none of the strategies could solve[10] (i.e., on which they ran for the full 10 s) and measured how much time was spent (on average) on interacting with the passive clause container (this includes insertions, deletions and the popping of the selected clauses). This average time is reported in the last column, from which we can see that, indeed, the more complex combined strategies are more expensive to execute.

Additionally, for comparison, the #queues column in the table reminds us how many "layer one" clause queues each strategy maintains (recall Section 6.1.3 and the number of "horizontal" splits each heuristic uses, as shown in Table 6.1). The multiplier (*2) corresponding to `av` and `pl` is rendered in brackets, because these two heuristics use the disjoint split mode. This means that when deciding on $dist_{AV}$ (or $dist_{Horn}$), each clause is strictly inserted only into one of two possible sub-containers (rather than possibly to more than one, as with the monotone mode). Correspondingly, the strategies are clearly separated into four groups in terms of average interaction time, where the monotone splits of `sl` and `th` are the costly ones (and maintaining the 4 queues of `th` costs more than the 3 queues of `sl`) whereas the disjoint split of the other two heuristics does not seem to be adding any measurable overhead.

We remark that the reported average times are not directly proportional to the speed of clause processing as each run was terminated after 10 s no matter how many selections were performed. Moreover, quite different search spaces could have been traversed by each of the strategies.

[10] There were 7808 such problems.

CHAPTER 7

Subsumption Demodulation in Superposition-based Theorem Proving

7.1 Introduction

For the efficiency of organizing proof search during saturation-based first-order theorem proving, simplification inference rules are of critical importance. These rules do not add new formulas to the search space, but simplify formulas by deleting (redundant) clauses from the search space. As such, simplification rules reduce the size of the search space and are crucial in making automated reasoning efficient.

When reasoning about properties of first-order logic with equality, one of the most common simplification rules is demodulation [KV13] for rewriting (and hence simplifying) formulas using unit equalities $l \simeq r$, where l and r are terms and $\simeq$ denotes equality. As a special case of superposition, demodulation is implemented in first-order provers such as E [SCV19], Spass [WDF+09], and Vampire [KV13].

The trace logic domain, introduced in Chapter 2, demands, however, new and efficient extensions of demodulation to reason about and simplify upon conditional equalities $C \rightarrow l \simeq r$, where C is a first-order formula. Such conditional equalities arise, for example, from the axiomatic semantics (introduced in Section 2.4) of software programs expressed in the standard while-like language $\mathcal{W}$ (described in Section 2.1), with C denoting a branching condition (such as a loop condition) and $l \simeq r$ encoding equational properties over program variables. We illustrate the need for generalized versions of demodulation in the following example.

7.1. Example Consider the following formulas expressed in the combined theory $\mathbb{D} \cup \mathbb{I}$ of difference logic and linear arithmetic:

$$\begin{array}{l} f(i) \simeq g(i) \\ 0 \leq i < n \rightarrow P(f(i)) \end{array} \tag{7.1}$$

Here, i is an implicitly universally quantified logical variable of sort $\mathbb{D}$, and n is a nullary function of sort $\mathbb{D}$. First-order reasoners will first clausify formulas (7.1), deriving:

$$\begin{array}{l} f(i) \simeq g(i) \\ 0 \not\leq i \vee i \not< n \vee P(f(i)) \end{array} \tag{7.2}$$

By applying demodulation over (7.2), the formula $0 \not\leq i \vee i \not< n \vee P(f(i))$ is rewritten[1] using the unit equality $f(i) \simeq g(i)$, yielding the clause $0 \not\leq i \vee i \not< n \vee P(g(i))$. That is, $0 \leq i < n \rightarrow P(g(i))$ is derived from (7.1) by one application of demodulation.

Let us now consider a slightly modified version of (7.1), as below:

$$\begin{array}{l} 0 \leq i < n \rightarrow f(i) \simeq g(i) \\ 0 \leq i < n \rightarrow P(f(i)) \end{array} \tag{7.3}$$

whose clausal representation is given by:

$$\begin{array}{l} 0 \not\leq i \vee i \not< n \vee f(i) \simeq g(i) \\ 0 \not\leq i \vee i \not< n \vee P(f(i)) \end{array} \tag{7.4}$$

It is again obvious that from (7.3) one can derive the formula $0 \leq i < n \rightarrow P(g(i))$, or equivalently the clause:

$$0 \not\leq i \vee i \not< n \vee P(g(i)) \tag{7.5}$$

Yet, *one cannot anymore apply demodulation-based simplification over* (7.4) *to derive such a clause*, as (7.4) contains no unit equality. □

In this chapter, we propose a generalized version of demodulation, called *subsumption demodulation*, allowing to rewrite terms and simplify formulas using rewriting based on conditional equalities, such as in (7.3). To do so, we extend demodulation with subsumption, that is, with deciding whether (an instance of a) clause C is a submultiset of a clause D. In particular, the non-equality literals of the conditional equality (i.e., the condition) need to subsume the unchanged literals of the simplified clause. This way, subsumption demodulation can be applied to non-unit clauses and is not restricted to have at least one premise clause that is a unit equality. We show that subsumption demodulation is a simplification rule of the superposition framework (Section 7.2), allowing for example to derive the clause (7.5) from (7.4) in one inference step. By properly adjusting clause indexing and multi-literal matching in first-order theorem provers, we provide an efficient implementation of subsumption demodulation in Vampire (Section 7.3). We conclude this chapter with an experimental evaluation of our work against state-of-the-art reasoners, including E [SCV19], Spass [WDF+09], CVC4 [BCD+11], and Z3 [DMB08], on the general domains of TPTP [Sut07] and SMT-LIB [BFT16] (Section 7.4).

[1] Assuming that g is simpler/smaller than f.

7.2 Subsumption Demodulation

In this section, we introduce a new simplification rule called subsumption demodulation, by extending demodulation to a simplification rule over conditional equalities. We do so by combining demodulation with subsumption checks to find simplifying applications of rewriting by non-unit (and hence conditional) equalities.

7.2.1 Subsumption Demodulation for Conditional Rewriting

Our rule of subsumption demodulation is defined below.

7.2. Definition (Subsumption Demodulation) *Subsumption demodulation* is the inference rule:

$$\frac{l \simeq r \vee C \qquad L[t] \vee D}{L[r\sigma] \vee D} \tag{7.6}$$

while requiring the following side conditions:

$$l\sigma = t \tag{7.7}$$
$$C\sigma \subseteq_M D \tag{7.8}$$
$$l\sigma \succ r\sigma \tag{7.9}$$
$$L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma \tag{7.10}$$

We call the equality $l \simeq r$ in the left premise of (7.6) the *rewriting equality* of subsumption demodulation.

Intuitively, the side conditions (7.7) and (7.8) of Definition 7.2 ensure the soundness of the rule: It is easy to see that if $l \simeq r \vee C$ and $L[t] \vee D$ are true, then $L[r\sigma] \vee D$ also holds. We thus conclude:

7.3. Theorem (Soundness) Subsumption demodulation is sound.

On the other hand, side conditions (7.9) and (7.10) of Definition 7.2 are vital to ensure that subsumption demodulation is a simplification rule (details follow in Subsection 7.2.2).

Detecting possible applications of subsumption demodulation involves (i) selecting one equality of the side clause as rewriting equality and (ii) matching each of the remaining literals, denoted C in (7.6), to some literal in the main clause. Step (i) is similar to finding unit equalities in demodulation, whereas step (ii) reduces to showing that C subsumes parts of the main premise. Informally speaking, subsumption demodulation combines demodulation and subsumption, as discussed in Section 7.3. Note that in step (ii), matching allows any instantiation of C to $C\sigma$ via substitution σ; yet, we do *not* unify the side and main premises of subsumption demodulation, as illustrated later in Example 7.6. Furthermore, we need to find a term t in the unmatched part $D \setminus C\sigma$ of

the main premise, such that t can be rewritten according to the rewriting equality into $r\sigma$.

As the ordering $\succ$ is partial, the conditions of Definition 7.2 must be checked a posteriori, that is after subsumption demodulation has been applied with a fixed substitution and revise the substitution if needed. Note however that if $l \succ r$ in the rewriting equality, then $l\sigma \succ r\sigma$ for any substitution, so checking the ordering a priori helps, as illustrated in the following example.

7.4. Example Let us consider the following two clauses:

$$C_1 = f(g(x)) \simeq g(x) \vee Q(x) \vee R(y)$$
$$C_2 = P(f(g(c))) \vee Q(c) \vee Q(d) \vee R(f(g(d)))$$

By the subterm property of $\succ$, we conclude that $f(g(x)) \succ g(x)$. Hence, the rewriting equality, as well as any instance of it, is oriented.

Let σ be the substitution $\sigma = \{x \mapsto c, y \mapsto f(g(d))\}$. Due to the previous paragraph, we know $f(g(c)) \succ g(c)$. As equalities are smaller than non-equality ones, we also conclude $P(f(g(c))) \succ f(g(c)) \simeq g(c)$. Thus, we have $P(f(g(c))) \vee Q(c) \vee Q(d) \vee R(f(g(d))) \succ f(g(c)) \simeq g(c) \vee Q(c) \vee R(f(g(d)))$ and we can apply subsumption demodulation to C_1 and C_2, deriving clause $C_3 = P(g(c)) \vee Q(c) \vee Q(d) \vee R(f(g(d)))$.

We note that demodulation cannot derive C_3 from C_1 and C_2, as there is no unit equality. □

Example 7.4 highlights the limitations of demodulation when compared to subsumption demodulation. Next, we illustrate different possible applications of subsumption demodulation using a fixed side premise and different main premises.

7.5. Example Consider the clause $C_1 = f(g(x)) \simeq g(y) \vee Q(x) \vee R(y)$. Only the first literal $f(g(x)) \simeq g(y)$ is a positive equality and as such eligible as rewriting equality. Note that $f(g(x))$ and $g(y)$ are incomparable w.r.t. $\succ$ due to occurrences of different variables, and hence whether $f(g(x))\sigma \succ g(y)\sigma$ depends on the chosen substitution σ.

(1) Consider the clause $C_2 = P(f(g(c))) \vee Q(c) \vee R(c)$ as the main premise. With the substitution $\sigma_1 = \{x \mapsto c, y \mapsto c\}$, we have $f(g(x))\sigma_1 \succ g(x)\sigma_1$ as $f(g(c)) \succ g(c)$ due to the subterm property of $\succ$, enabling a possible application of subsumption demodulation over C_1 and C_2.

(2) Consider now $C_3 = P(g(f(g(c)))) \vee Q(c) \vee R(f(g(c)))$ as the main premise and the substitution $\sigma_2 = \{x \mapsto c, y \mapsto f(g(c))\}$. We have $g(y)\sigma_2 \succ f(g(x))\sigma_2$, as $g(f(g(c)) \succ f(g(c))$. The instance of the rewriting equality is oriented differently in this case than in the previous one, enabling a possible application of subsumption demodulation over C_1 and C_3.

(3) On the other hand, using the clause $C_4 = P(f(g(c))) \vee Q(c) \vee R(z)$ as the main premise, the only substitution we can use is $\sigma_3 = \{x \mapsto c, y \mapsto z\}$. The corresponding

instance of the rewriting equality is then $f(g(c)) \simeq g(z)$, which cannot be oriented in general. Hence, subsumption demodulation cannot be applied in this case, even though we can find the matching term $f(g(c))$ in C_4. □

As mentioned before, the substitution σ appearing in subsumption demodulation can only be used to instantiate the side premise, but not for unifying side and main premises, as we would not obtain a simplification rule.

7.6. Example Consider the clauses:

$$C_1 = f(c) \simeq c \vee Q(d)$$
$$C_2 = P(f(c)) \vee Q(x)$$

As we cannot match $Q(d)$ to $Q(x)$ (although we could match $Q(x)$ to $Q(d)$), subsumption demodulation is not applicable with premises C_1 and C_2. □

7.2.2 Simplification using Subsumption Demodulation

Note that in the special case where C is the empty clause in (7.6), subsumption demodulation reduces to demodulation and hence it is a simplification rule. We next show that this is the case in general:

7.7. Theorem (Simplification Rule) Subsumption demodulation is a simplification rule, and can therefore be written as

$$\frac{l \simeq r \vee C \qquad \cancel{L[t] \vee D}}{L[r\sigma] \vee D}$$

while requiring the following side conditions:

$$l\sigma = t \tag{7.11}$$
$$C\sigma \subseteq_M D \tag{7.12}$$
$$l\sigma \succ r\sigma \tag{7.13}$$
$$L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma \tag{7.14}$$

Proof. Because of the second condition of the definition of subsumption demodulation, $L[t] \vee D$ is clearly a logical consequence of $L[r\sigma] \vee D$ and $l \simeq r \vee C$. Moreover, from the fourth condition, we trivially have $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$. It thus remains to show that $L[r\sigma] \vee D$ is smaller than $L[t] \vee D$ w.r.t. $\succ$. As $t = l\sigma \succ r\sigma$, the monotonicity property of $\succ$ asserts that $L[t] \succ L[r\sigma]$, and hence $L[t] \vee D \succ L[r\sigma] \vee D$. This concludes that $L[t] \vee D$ is redundant w.r.t. the conclusion and left-most premise of subsumption demodulation. □

7.8. Example By revisiting Example 7.4, Theorem 7.7 asserts that clause C_2 is simplified into C_3, and subsumption demodulation deletes C_2 from the search space. □

7.2.3 Refining Redundancy

The fourth condition defining subsumption demodulation in Definition 7.2 is required to ensure that the main premise of subsumption demodulation becomes redundant. However, comparing clauses w.r.t. the ordering $\succ$ is computationally expensive; yet, not necessary for subsumption demodulation. Following the notation of Definition 7.2, let D' such that $D = C\sigma \vee D'$. By properties of multiset orderings, the condition $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$ is equivalent to $L[t] \vee D' \succ (l \simeq r)\sigma$, as the literals in $C\sigma$ occur on both sides of $\succ$. This means, to ensure the redundancy of the main premise of subsumption demodulation, we only need to ensure that there is a literal from $L[t] \vee D$ such that this literal is bigger that the rewriting equality.

7.9. Theorem (Refining Redundancy) The following conditions are equivalent:

$$L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma \tag{7.15}$$
$$L[t] \vee D' \succ (l \simeq r)\sigma \tag{7.16}$$

As mentioned in Subsection 7.2.1, the application of subsumption demodulation involves checking that an ordering condition between premises holds (side condition (7.10) in Definition 7.2). Theorem 7.9 asserts that we only need to find a literal in $L[t] \vee D'$ that is bigger than the rewriting equality to ensure that the ordering condition is fulfilled. In the next section we show that by re-using and properly changing the underlying machinery of first-order provers for demodulation and subsumption, subsumption demodulation can efficiently be implemented in superposition-based proof search.

7.3 Subsumption Demodulation in Vampire

We implemented subsumption demodulation in the first-order theorem prover Vampire. Our implementation consists of about 5000 lines of C++ code and is part of Vampire[2] starting from version 4.5.

As for any simplification rule, we implemented the forward and backward versions of subsumption demodulation separately. Our new Vampire options controlling subsumption demodulation are `-fsd` and `-bsd`, both with possible values `on` and `off`, to respectively enable forward and backward subsumption demodulation.

As discussed in Section 7.2, subsumption demodulation uses reasoning based on a combination of demodulation and subsumption. Algorithm 2 details our implementation for *forward subsumption demodulation*. In a nutshell, given a clause D as main premise, (forward) subsumption demodulation in Vampire consists of the following main steps:

1. *Retrieve candidate clauses* C as side premises of subsumption demodulation (line 2 of Algorithm 2). To this end, we design a new clause index with imperfect filtering, by modifying the subsumption index in Vampire, as discussed later in this section.

[2] Available at https://github.com/vprover/vampire.

Algorithm 2 Forward Subsumption Demodulation – FSD.

Input: Clause D, to be used as main premise
Output: Simplified clause D' if (forward) subsumption demodulation is possible

1: // Retrieve candidate side premises
2: $candidates := FSDIndex.Retrieve(D)$
3: **for each** $C \in candidates$ **do**
4: **while** $m := FindNextMLMatch(C, D)$ **do**
5: $\sigma' := m.GetSubstitution()$
6: $E := m.GetRewritingEquality()$
7: // E is of the form $l \simeq r$, for some terms l, r
8: **if** exists term t in $D \setminus C\sigma'$ and substitution $\sigma \supseteq \sigma'$ s.t. $t = l\sigma$ **then**
9: **if** $CheckOrderingConditions(D, E, t, \sigma)$ **then**
10: $D' := BuildSimplifiedClause(D, E, t, \sigma)$ **return** D'

2. *Prune candidate clauses* by checking the conditions of subsumption demodulation (lines 4–9 of Algorithm 2), in particular selecting a rewriting equality and matching the remaining literals of the side premise to literals of the main premise. After this, prune further by performing a posteriori checks for orienting the rewriting equality E, and checking the redundancy of the given main premise D. To do so, we revised multi-literal matching and redundancy checking in Vampire (see later).
3. *Build the simplified clause* by simplifying and deleting the (main) premise D of subsumption demodulation using (forward) simplification (line 10 of Algorithm 2).

Our implementation of *backward subsumption demodulation* requires only a few changes to Algorithm 2: (i) we use the input clause as side premise C of backward subsumption demodulation and (ii) we retrieve candidate clauses D as potential main premises of subsumption demodulation. Additionally, (iii) instead of returning a single simplified clause D', we record a replacement clause for each candidate clause D where a simplification was possible.

Clause Indexing for Subsumption Demodulation. We build upon the indexing approach [SRV01] used for subsumption in Vampire, where a subsumption index stores and retrieves candidate clauses for subsumption. Each clause is indexed by exactly one of its literals. In principle, any literal of the clause can be chosen. To reduce the number of retrieved candidates, the best literal is chosen in the sense that the chosen literal maximizes a certain heuristic (e.g., maximal weight). Since the subsumption index is not a perfect index (i.e., it may retrieve non-subsumed clauses), additional checks on the retrieved clauses are performed.

Using the subsumption index of Vampire as the clause index for forward subsumption demodulation would however omit to retrieve clauses (side premises) in which the rewriting equality is chosen as key for the index, omitting this way a possible application of subsumption demodulation. Hence, we need a new clause index in which the best literal

can be adjusted to be the rewriting equality. To address this issue, we added a new clause index, called the *forward subsumption demodulation index (FSD index)*, to Vampire, as follows. We index potential side premises either by their best literal (according to the heuristic), the second-best literal, or both. If the best literal in a clause C is a positive equality (i.e., a candidate rewriting equality) but the second-best is not, C is indexed by the second-best literal, and vice versa. If both the best and the second-best literal are positive equalities, C is indexed by both of them. Furthermore, because the FSD index is exclusively used by forward subsumption demodulation, this index only needs to keep track of clauses that contain at least one positive equality.

In the backward case, we reuse Vampire's index for backward subsumption. Analogously to the forward case, we need to query the index by the best literal, the second-best literal, or both.

Multi-literal Matching. Similarly to the subsumption index, our new subsumption demodulation index is not a perfect index, that is, it performs imperfect filtering for retrieving clauses. Therefore, additional post-checks are required on the retrieved clauses. In our work, we devised a multi-literal matching approach to (i) choose the rewriting equality among the literals of the side premise C, and (ii) check whether the remaining literals of C can be uniformly instantiated to the literals of the main premise D of subsumption demodulation. There are multiple ways to organize this process. A simple approach is to (i) first pick any equality of a side premise C as the rewriting equality of subsumption demodulation, and then (ii) invoke the existing multi-literal matching machinery of Vampire to match the remaining literals of C with a subset of literals of D. For the latter step (ii), the task is to find a substitution σ such that $C\sigma$ becomes a submultiset of the given clause D. If the choice of the rewriting equality in step (i) turns out to be wrong, we backtrack. In our work, we revised the existing multi-literal matching machinery of Vampire to a new multi-literal matching approach for subsumption demodulation, by using the steps (i)-(ii) and interleaving equality selection with matching.

We note that the substitution σ in step (ii) above is built in two stages: first we get a partial substitution σ' from multi-literal matching and then (possibly) extend σ' to σ by matching term instances of the rewriting equality with terms of $D \setminus C\sigma$.

7.10. Example Let D be the clause $P(f(c,d)) \vee Q(c)$. Assume that our (FSD) clause index retrieves the clause $C = f(x,y) \simeq y \vee Q(x)$ from the search space (line 2 of Algorithm 2). We then invoke our multi-literal matcher (line 4 of Algorithm 2), which matches the literal $Q(x)$ of C to the literal $Q(c)$ of D and selects the equality literal $f(x,y) \simeq y$ of C as the rewriting equality for subsumption demodulation over C and D. The matcher returns the choice of rewriting equality and the partial substitution $\sigma' = \{x \mapsto c\}$. We arrive at the final substitution $\sigma = \{x \mapsto c, y \mapsto d\}$ only when we match the instance $f(x,y)\sigma'$, that is $f(c,y)$, of the left-hand side of the rewriting equality to the literal $f(c,d)$ of D. Using σ, subsumption demodulation over C and D will derive $P(d) \vee Q(c)$, after ensuring that D becomes redundant (line 10 of Algorithm 2). □

We further note that multi-literal matching is an NP-complete problem. Our multi-literal matching problems may have more than one solution, with possibly only some (or none) of them leading to successful applications of subsumption demodulation. In our implementation, we examine all solutions retrieved by multi-literal matching. We also experimented with limiting the number of matches examined after multi-literal matching but did not observe relevant improvements. Yet, our implementation in Vampire also supports an additional option allowing the user to specify an upper bound on how many solutions of multi-literal matching should be examined.

Redundancy Checking. To ensure redundancy of the main premise D after the subsumption demodulation inference, we need to check two properties. First, the instance $E\sigma$ of the rewriting equality E must be oriented. This is a simple ordering check. Second, the main premise D must be larger than the side premise C. Thanks to Theorem 7.9, this latter condition is reduced to finding a literal among the unmatched part of the main premise D that is bigger than the instance $E\sigma$ of the rewriting equality E.

7.11. Example In the case of Example 7.10, the rewriting equality E is oriented and hence $E\sigma$ is also oriented. Next, the literal $P(f(c,d))$ is bigger than $E\sigma$, and hence D is redundant w.r.t. C and D'. □

7.4 Experiments

Similar to layered clause selection, the development of subsumption demodulation has initially been motivated by the trace logic domain. We will later empirically investigate the impact of subsumption demodulation on reasoning in the trace logic domain, using an experimental evaluation of subsumption demodulation on the trace logic domain as part of the broader experimental evaluation in Chapter 9.

In this section, will evaluate our implementation of subsumption demodulation in Vampire on the general domains of the TPTP [Sut17] and SMT-LIB [BFT16] repositories.

Benchmark Setup. All our experiments were carried out on the StarExec cluster [SST14]. From the 22,686 problems in the TPTP benchmark set (version 7.3.0), Vampire can parse 18,232 problems.[3] Out of these problems, we only used those problems that involve equalities as subsumption demodulation is only applicable in the presence of (at least one) equality. As such, we used 13,924 TPTP problems in our experiments. On the other hand, when using the SMT-LIB repository (release 2019-05-06), we chose the benchmarks from categories LIA, UF, UFDT, UFDTLIA, and UFLIA, as these benchmarks involve reasoning with both theories and quantifiers and the background theories are the theories that Vampire supports. These are 22,951 SMT-LIB problems in total, of which 22,833 problems remain after removing those where equality does not occur.

[3]The other problems contain features, such as higher-order logic, that have not been implemented in Vampire yet.

Table 7.1: Comparing Vampire with and without subsumption demodulation on TPTP, using Vampire in portfolio mode.

Configuration	Total	Solved	New (SAT+UNSAT)
Vampire	13,924	9,923	–
Vampire, with FSD	13,924	9,757	20 (3+17)
Vampire, with BSD	13,924	9,797	14 (2+12)
Vampire, with FSD and BSD	13,924	9,734	30 (6+24)

Table 7.2: Comparing Vampire with and without subsumption demodulation on SMT-LIB, using Vampire in portfolio mode.

Configuration	Total	Solved	New (SAT+UNSAT)
Vampire	22,833	13,705	–
Vampire, with FSD	22,833	13,620	55 (1+54)
Vampire, with BSD	22,833	13,632	48 (0+48)
Vampire, with FSD and BSD	22,833	13,607	76 (0+76)

Comparative Experiments with Vampire. As a first experimental study, we compared the performance of subsumption demodulation in Vampire for different values of `-fsd` and `-bsd`, that is by using forward (FSD) and/or backward (BSD) subsumption demodulation. To this end, we evaluated subsumption demodulation using the CASC and SMTCOMP schedules of Vampire's portfolio mode. To test subsumption demodulation with the portfolio mode, we added the options `-fsd on` and/or `-bsd on` to *all* strategies of Vampire. While the resulting strategy schedules could potentially be further improved, it allowed us to test FSD/BSD with a variety of strategies.

Our results are summarized in Tables 7.1-7.2. The first column of these tables lists the Vampire version and configuration, where Vampire refers to Vampire in its portfolio mode (version 4.4). Lines 2-4 of these tables use our new Vampire, that is our implementation of subsumption demodulation in Vampire. The column "Solved" reports, respectively, the total number of TPTP and SMT-LIB problems solved by the considered Vampire configurations. Column "New" lists, respectively, the number of TPTP and SMT-LIB problems solved by the version with subsumption demodulation but not by the portfolio version of Vampire. This column also indicates in parentheses how many of the solved problems were satisfiable/unsatisfiable.

While in total the portfolio mode of Vampire can solve more problems, we note that this comes at no surprise as the portfolio mode of Vampire is highly tuned using the existing Vampire options. In our experiments, we were interested to see whether subsumption demodulation in Vampire can solve problems that cannot be solved by the portfolio mode of Vampire. In future work, the portfolio mode should be tuned by also taking into account subsumption demodulation, which then ideally leads to an overall increase in performance. The columns "New" of Tables 7.1-7.2 give indeed practical evidence of

Table 7.3: Comparing Vampire with subsumption demodulation against other solvers, using the "new" TPTP and SMT-LIB problems of Tables 7.1-7.2 and running Vampire in portfolio mode.

Solver/configuration	TPTP problems	SMT-LIB problems
Baseline: Vampire, with FSD and BSD	30	76
E with `--auto-schedule`	14	-
Spass (default)	4	-
Spass (local contextual rewriting)	6	-
Spass (subterm contextual rewriting)	5	-
CVC4 (default)	7	66
Z3 (default)	-	49
Only solved by Vampire, with FSD and BSD	11	0

the impact of subsumption demodulation: There are 30 new TPTP problems and 76 SMT-LIB problems[4] that the portfolio version of Vampire cannot solve, but forward and backward subsumption demodulation in Vampire can.

New Problems Solved Only by Subsumption Demodulation. Building upon our results from Tables 7.1-7.2, we analyzed how many new problems subsumption demodulation in Vampire can solve when compared to other state-of-the-art reasoners. To this end, we evaluated our work against the superposition provers E (version 2.4) and Spass (version 3.9), as well as the SMT solvers CVC4 (version 1.7) and Z3 (version 4.8.7). We note however, that when using our 30 new problems from Table 7.1, we could not compare our results against Z3 as Z3 does not natively parse TPTP. On the other hand, when using our 76 new problems from Table 7.2, we only compared against CVC4 and Z3, as E and Spass do not support the SMT-LIB syntax.

Table 7.3 summarizes our findings. First, 11 of our 30 "new" TPTP problems can only be solved using forward and backward subsumption demodulation in Vampire; none of the other systems were able to solve these problems.

Second, while all our 76 "new" SMT-LIB problems can also be solved by CVC4 and Z3 together, we note that out of these 76 problems there are 10 problems that CVC4 cannot solve, and similarly 27 problems that Z3 cannot solve.

Comparative Experiments without AVATAR. Finally, we investigated the effect of subsumption demodulation in Vampire without Avatar [Vor14]. We used the default mode of Vampire (that is, without using a portfolio approach) and turned off the Avatar setting. While this configuration solves fewer problems than the portfolio mode of Vampire, so far Vampire is the only superposition-based theorem prover implementing Avatar. Hence, evaluating subsumption demodulation in Vampire without Avatar is more relevant to other reasoners. Further, as Avatar may often split non-unit clauses

[4]The list of these new problems is available at
https://gist.github.com/JakobR/605a7b7db0101259052e137ade54b32c.

Table 7.4: Comparing Vampire in default mode and without Avatar, with and without subsumption demodulation.

	TPTP problems			SMT-LIB problems		
Configuration	Total	Solved	New (SAT+UNSAT)	Total	Solved	New (SAT+UNSAT)
Vampire	13,924	6,601	–	22,833	9,608	–
Vampire (FSD)	13,924	6,539	152 (13+139)	22,833	9,597	134 (1+133)
Vampire (BSD)	13,924	6,471	112 (12+100)	22,833	9,541	87 (0+87)
Vampire (FSD+BSD)	13,924	6,510	190 (15+175)	22,833	9,581	173 (1+172)

into unit clauses, it may potentially simulate applications of subsumption demodulation using demodulation. Table 7.4 shows that this is indeed the case: With both FSD and BSD enabled, subsumption demodulation in Vampire can prove 190 TPTP problems and 173 SMT-LIB examples that the default Vampire without Avatar cannot solve. Again, the column "New" denotes the number of problems solved by the respective configuration but not by the default mode of Vampire without Avatar.

7.5 Related Work

While several approaches generalize demodulation in superposition-based theorem proving, we argue that subsumption demodulation improves existing methods either in terms of applicability and/or efficiency. The Avatar architecture of first-order provers [Vor14] splits general clauses into components with disjoint sets of variables, potentially enabling demodulation inferences whenever some of these components become unit equalities. Example 7.1 demonstrates that subsumption demodulation applies in situations where Avatar does not. In each clause of (7.4), all literals share the variable i, and hence none of the clauses from (7.4) can be split using Avatar. That is, Avatar would not generate unit equalities from (7.4), and therefore cannot apply demodulation over (7.4) to derive (7.5).

The local rewriting approach of [Wei01] requires rewriting equality literals to be maximal[5] in clauses. However, following [KV13], for efficiency reasons we consider equality literals to be "smaller" than non-equality literals. In particular, the equality literals of clauses (7.4) are "smaller" than the non-equality literals, preventing thus the application of local rewriting in Example 7.1.

To the extent of our knowledge, the ordering restrictions on non-unit rewriting [Wei01] do not ensure redundancy, and thus the rule is not a simplification inference rule. Subsumption demodulation includes all necessary conditions and we prove it to be a simplification rule. Furthermore, we show how the ordering restrictions can be simplified which en-

[5] With respect to the clause ordering.

ables an efficient implementation, and then explain how such an implementation can be realized.

We further note that the contextual rewriting rule of [BG94] is more general than our rule of subsumption demodulation, and has been first implemented in the Saturate system [NN93]. Yet, efficiently automating contextual rewriting is extremely challenging, while subsumption demodulation requires no radical changes in the existing machinery of superposition provers (see Section 7.3).

To the best of our knowledge, except Spass [WDF+09] and Saturate, no other state-of-the-art superposition provers implement variants of conditional rewriting. Subterm contextual rewriting [WW08] is a refined notion of contextual rewriting and is implemented in Spass. A major difference of subterm contextual rewriting when compared to subsumption demodulation is that in subsumption demodulation the discovery of the substitution is driven by the side conditions whereas in subterm contextual rewriting the side conditions are evaluated by checking the validity of certain implications by means of a reduction calculus. This reduction calculus recursively applies another restriction of contextual rewriting called recursive contextual ground rewriting, among other standard reduction rules. While subterm contextual rewriting is more general, we believe that the benefit of subsumption demodulation comes with its relatively easy and efficient integration within existing superposition reasoners, as evidenced also in Section 7.4.

Local contextual rewriting [HPWW13] is another refinement of contextual rewriting implemented in Spass. In our experiments, it performed similarly to subterm contextual rewriting.

Finally, we note that SMT-based reasoners also implement various methods to efficiently handle conditional equalities [RWB+17, BGMR15]. Yet, the setting is very different as they rely on the DPLL(T) framework [GHN+04] rather than implementing superposition.

CHAPTER 8

Interactive Visualization of Saturation Attempts in Vampire

8.1 Introduction

The performance of saturation-based theorem provers crucially depends on the logical representation of its input problem and the deployed reasoning strategies during proof search. As such, users and developers of saturation-based theorem provers, and automated reasoners in general, typically face the burden of analyzing (failed) proof attempts produced by the prover, with the ultimate goal to refine the input and/or proof strategies making the prover succeed in proving its input. Understanding (some of) the reasons why the prover failed is however very hard and requires a considerable amount of work by highly qualified experts in theorem proving, hindering thus the use of theorem provers in many application domains.

In this chapter, we address this challenge and *introduce the* SatVis *tool to ease the task of analyzing failed proof attempts in saturation-based reasoning.* We designed SatVis to support interactive visualization of the saturation algorithm used in Vampire, with the goal to ease the manual analysis of Vampire proofs as well as failed proof attempts in Vampire. Inputs to SatVis are proof (attempts) produced by Vampire. Our tool consists of (i) an explicit visualization of the DAG-structure of the saturation proof (attempt) of Vampire and (ii) interactive transformations of the DAG for pruning and reformatting the proof (attempt). In its current setting, SatVis can be used only in the context of Vampire. Yet, by parsing/translating proofs (or proof attempts) of other provers into the Vampire proof format, SatVis can be used in conjunction with other provers as well.

When feeding Vampire proofs to SatVis, SatVis supports both users and developers of Vampire to understand and refactor Vampire proofs, and to manually proof check

soundness of Vampire proofs. When using SatVis on failed proof attempts of Vampire, SatVis supports users and developers of Vampire to analyze how Vampire explored its search space during proof search, that is, to understand which clauses were derived and why certain clauses have not been derived at various steps during saturation. By doing so, the SatVis proof visualization framework gives valuable insights on how to revise the input problem encoding of Vampire and/or implement domain-specific optimizations in Vampire. We therefore believe that SatVis improves the state-of-the-art in the use and applications of theorem proving at least in the following scenarios: (i) helping Vampire developers to debug and further improve Vampire, (ii) helping Vampire users to tune Vampire to their applications, by not treating Vampire as a black-box but by understanding and using its appropriate proof search options; and (iii) helping inexperienced users in saturation-based theorem proving to learn using Vampire and first-order proving in general.

Contributions. The contribution of this paper comes with the design of the SatVis tool for analyzing proofs, as well as proof attempts of the Vampire theorem prover. SatVis is available at:

https://github.com/gleiss/saturation-visualization.

We discuss the challenges we faced for analyzing proof attempts of Vampire (Section 8.2), describe implementation-level details of SatVis 1.0 (Section 8.3), and overview related work (Section 8.4)

8.2 Analysis of Saturation Attempts of Vampire

We now discuss how to efficiently analyze saturation attempts of Vampire in SatVis.

Analyzing saturation attempts. To understand saturation (attempts), we have to analyze the generating inferences performed during saturation (attempts).

On the one hand, we are interested in the *useful* clauses, that is, the derived and activated clauses that are part of the proof we expect Vampire to find. In particular, we check whether these clauses occur in *Active*. (i) If this is the case for a given useful clause (or a simplified variant of it), we are done with processing this useful clause and optionally check the derivation of that clause against the expected derivation. (ii) If not, we have to identify the reason why the clause was not added to *Active*, which can either be the case because (ii.a) the clause (or a simplified version of it) was never chosen from *Passive* to be activated or (ii.b) the clause was not even added to *Passive*. In case (ii.a), we investigate why the clause was not activated. This involves checking which simplified version of the clause was added to *Passive* and checking the value of clause selection in Vampire on that clause. In case (ii.b), it is needed to understand why the clause was not added to *Passive*, that is, why no generating inference between suitable premise clauses was performed. This could for instance be the case because one of the premises

```
...
[SA] passive: 160. v = a(l11(s(nl8)),$sum(i(main_end),1)) [superposition 70,118]
[SA] active: 163. i(main_end) != -1 [term algebras distinctness 162]
[SA] active: 92. ~'Sub'(X5,p(X4)) | 'Sub'(X5,X4) | zero = X4 [superposition 66,44]
[SA] new: 164. 'Sub'(p(p(X0)),X0) | zero = X0 | zero = p(X0) [resolution 92,94]
[SA] passive: 164. 'Sub'(p(p(X0)),X0) | zero = X0 | zero = p(X0) [resolution 92,94]
[SA] active: 132. v = a(l11(s(s(zero))),2) [superposition 70,124]
[SA] new: 165. v = a(l8(s(s(zero))),2) | i(l8(s(s(zero)))) = 2 [superposition 132,72]
[SA] new: 166. v = a(l8(s(s(zero))),2) | i(l8(s(s(zero)))) = 2 [superposition 72,132]
[SA] active: 90. s(X1) != X0 | p(X0) = X1 | zero = X0 [superposition 22,44]
[SA] new: 167. X0 != X1 | p(X0) = p(X1) | zero = X1 | zero = X0 [superposition 90,44]
[SA] new: 168. p(s(X0)) = X0 | zero = s(X0) [equality resolution 90]
[SA] new: 169. p(s(X0)) = X0 [term algebras distinctness 168]
...
```

Figure 8.1: Screenshot of a saturation attempt of Vampire.

was not added to *Active*, in which case we recurse with the analysis on that premise, or because clause selection in Vampire prevented the inference.

On the other hand, we are interested in the *useless* clauses: that is, the clauses which were generated or even activated but are unrelated to the proof Vampire will find. These clauses often slow down the proof search by several magnitudes. It is therefore crucial to limit their generation or at least their activation. To identify the useless clauses that are activated, we need to analyze the set *Active*, whereas to identify the useless clauses, which are generated but never activated, we have to investigate the set *Passive*.

Saturation output. We now discuss how SatVis reconstructs the clause sets *Active* and *Passive* from a Vampire saturation (attempt). Vampire is able to log a list of events, where each event is classified as either (i) new C (ii) passive C, or (iii) active C, for a given clause C. The list of events produced by Vampire satisfies the following properties: (a) any clause is at most once newly created, added to *Passive* and added to *Active*; (b) if a clause is added to *Passive*, it was newly created in the same iteration, and (c) if a clause is added to *Active*, it was newly created and added to *Passive* at some point. Figure 8.1 shows a part of the output logged by Vampire while performing a saturation attempt (`SA`).

Starting from an empty derivation and two empty sets, the derivation graph and the sets *Active* and *Passive* corresponding to a given saturation attempt of Vampire are computed in SatVis by traversing the list of events produced by Vampire and iteratively changing the derivation and the sets *Active* and *Passive*, as follows:

(i) new C: add the new node C to the derivation and construct the edges (C_i, C), for any premise C_i of the inference deriving C. The sets *Active* or *Passive* remain unchanged;
(ii) passive C: add the node C to *Passive*. The derivation and *Active* remain unchanged;
(iii) active C: remove the node C from *Passive* and add it to *Active*. The derivation remains unchanged.

Interactive Visualization. The large number of inferences during saturation in Vampire makes the direct analysis of saturation attempts of Vampire impossible within a reasonable amount of time. To overcome this problem, in SatVis we *interactively* visualize the derivation graph of the Vampire saturation. The graph-based visualization of SatVis brings the following benefits:

• Navigating through the graph visualization of a Vampire derivation is easier for users rather than working with the Vampire derivation encoded as a list of hyperedges. In particular, both (i) navigating to the premises of a selected node/clause, and (ii) searching for inferences having a selected node/clause as premise is performed fast in SatVis.

• SatVis visualizes only the nodes/clauses that are part of a derivation of an activated clause, and in this way ignores uninteresting inferences.

• SatVis merges the preprocessing inferences, such that each clause resulting from preprocessing has as direct premise the input formula it is derived from.

Yet, a straightforward graph-based visualization of Vampire saturations in SatVis would bring the following practical limitations on using SatVis:

(i) displaying additional meta-information on graph nodes, such as the inference rule used to derive a node, is computationally very expensive, due to the large number of inferences used during saturation;

(ii) manual search for particular/already processed nodes in relatively large derivations would take too much time;

(iii) subderivations are often interleaved with other subderivations due to an imperfect automatic layout of the graph.

SatVis addresses the above challenges using its following interactive features:

- SatVis displays meta-information only for a selected node/clause;
- SatVis supports different ways to locate and select clauses, such as full-text search, search for direct children and premises of the currently selected clauses, and search for clauses whose derivation contains all currently selected nodes;
- SatVis supports transformations/fragmentations of derivations. In particular, it is possible to restrict and visualize the derivation containing only the clauses that form the derivation of a selected clause, or visualize only clauses whose derivation contains a selected clause.
- SatVis allows to (permanently) highlight one or more clauses in the derivation.

Figure 8.2 illustrates some of the above features of SatVis, using output from Vampire similar to Figure 8.1 as input to SatVis.

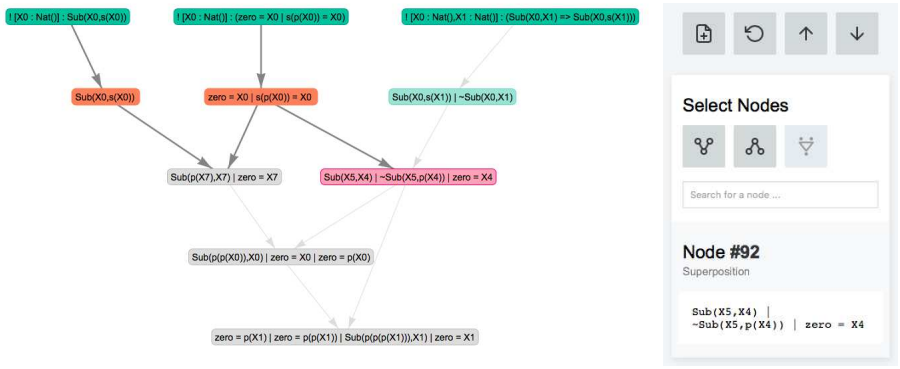

Figure 8.2: Screenshot of SatVis showing visualized derivation and interaction menu.

8.3 Implementation of SatVis 1.0

We implemented SatVis as a web application, allowing SatVis to be easily used on any platform. Written in Python3, SatVis contains about 2,200 lines of code. For the generation of graph layouts, we rely on `pygraphviz`[1], whereas graph/derivation visualizations are created with `vis.js`[2]. We experimented with SatVis on examples from the trace logic domain (see Chapters 2-4) using an Intel Core i5 3.1Ghz machine with 16 GB of RAM, allowing us to refine and successfully generate Vampire proofs for functional properties of software programs over (unbounded) arrays and loops.

SatVis workflow. SatVis takes as input a text file containing the output of a Vampire saturation attempt. An example of a partial input to SatVis is given in Figure 8.1. SatVis then generates a DAG representing the derivation of the considered Vampire saturation output, as presented in Section 8.2 and discussed later. Next, SatVis generates the graph layout of for the generated DAG, enriched with configured style information. Finally, SatVis renders and visualizes the Vampire derivation corresponding to its input, and allows interactive visualizations of its output, as discussed in Section 8.2 and detailed below.

DAG generation of saturation outputs. SatVis parses its input line by line using regex pattern matching in order to generate the nodes of the graph. Next, SatVis uses a post-order traversal algorithm to sanitize nodes and remove redundant ones. The result is then passed to `pygraphviz` to generate a graph layout. While `pygraphviz` finds layouts for thousands of nodes within less than three seconds, we would like to improve the scalability of the tool further. It would be beneficial to preprocess and render nodes incrementally, while ensuring stable layouts for SatVis graph transformations. We leave this engineering task for future work.

[1] https://pygraphviz.github.io.
[2] https://visjs.org/.

Interactive visualization The interactive features of SatVis support (i) various node searching mechanisms, (ii) graph transformations, and (iii) the display of meta-information about a specific node. We can efficiently search for nodes by (partial) clause, find parents or children of a node, and find common consequences of a number of nodes. Graph transformations in SatVis allow to only render a certain subset of nodes from the SatVis DAG, for example, displaying only transitive parents or children of a certain node.

8.4 Related Work

While standardizing the input format of automated reasoners is an active research topic, see e.g. the SMT-LIB [BFT17] and TPTP [Sut07] standards, coming up with an input standard for representing and analyzing proofs and proof attempts of automated reasoners has received so far very little attention. The TSTP library [Sut07] provides input/output standards for automated theorem proving systems. Yet, unlike SatVis, TSTP does not analyze proof attempts but only supports the examination of first-order proofs. We note that Vampire proofs (and proof attempts) contain first-order formulas with theories, which is not fully supported by TSTP.

Using a graph-layout framework, for instance Graphviz [GN00], it is relatively straightforward to visualize the DAG derivation graph induced by a saturation attempt of a first-order prover. For example, the theorem prover E [Sch02] is able to directly output its saturation attempt as an input file for Graphviz. The visualizations generated in this way are useful however only for analyzing small derivations with at most 100 inferences, but cannot practically be used to analyze and manipulate larger proof attempts. We note that it is quite common to have first-order proofs and proof attempts with more than 1,000 or even 10,000 inferences, especially in applications of theorem proving in software verification. In our SatVis framework, the interactive features of our tool allow one to analyze such large(r) proof attempts.

The framework [Rot16] eases the manual analysis of proof attempts in Z3 [DMB08] by visualizing quantifier instantiations, case splits, and conflicts. While both [Rot16] and SatVis are built for analyzing (failed) proof attempts, they target different architectures (SMT-solving resp. superposition-based proving) and therefore differ in their input format and in the information they visualize. The frameworks [BBE+09, LRR14] visualize proofs derived in a natural deduction/sequent calculus. Unlike these approaches, SatVis targets clausal derivations generated by saturation-based provers using the superposition inference system. As a consequence, our tool can be used to focus only on the clauses that have been actively used during proof search, instead of having to visualize the entire set of clauses, including clauses unused during proof search. We finally note that proof checkers, such as DRAT-trim [WHH14], support the soundness analysis of each inference step of a proof, and do not focus on failing proof attempts nor do they visualize proofs.

CHAPTER 9

Experiments

In the first part of the thesis (chapters 2–4), we formalized the correctness of software program properties in trace logic, and in the second part of the thesis (chapters 5–8), we described how to reason about the resulting formalizations with the superposition-based theorem prover Vampire. The combination of these two parts yields a fully automatic approach to establish the correctness of software program properties. In this chapter, we demonstrate the effectiveness of this approach, using an experimental evaluation on a large number of benchmarks.

We will first discuss in Section 9.1 the two sets of benchmarks we use in our experiments. Then, we will describe in Section 9.2 the tool Rapid, which implements the ideas from chapters 2–4. Afterwards, we will discuss in Section 9.3 the custom version of Vampire used for the experiments, obtained by extending Vampire with further optimizations specific to the trace-logic domain. Finally, we will present an experimental evaluation in Section 9.4.

9.1 Benchmarks

Arrays. We consider a first set of benchmarks arrays[1], which focuses on functional properties over programs including arrays and loops, and contains 45 programs with a total of 103 properties. It was obtained as follows. We started from interesting and challenging Java- and C-like verification examples from the SV-Comp benchmark repository [Bey20] and manually transformed them into our input language. Note that SV-Comp benchmarks encode properties featuring universal quantification by extending the corresponding program with an additional loop containing a standard C-like assertion. For instance, the property

$$\forall i^{\mathbb{I}}.\ 0 \leq i < alength \rightarrow P(a(l_{end}, i)).$$

[1] Available at https://github.com/gleiss/rapid/examples/arrays.

```
1  func main()
2  {
3    const Int[] a;
4    const Int[] b;
5    Int[] c;
6    const Int length;
7
8    Int i = 0;
9    while(i < length)
10   {
11     c[i] = a[i] + b[i];
12     i = i + 1;
13   }
14 }
```

(a) Pointwise addition.

```
1  func main()
2  {
3    Int[] a;
4    const Int alength;
5
6    Int i = 0;
7    while (i < alength)
8    {
9      a[i] = a[i] + 1;
10     i = i + 1;
11   }
12 }
```

(b) Increment each position.

Figure 9.1: Examples from ARRAYS.

would be encoded by extending the program with a loop

```
for(int i = 0; i < alength; i++){ assert(P(a[i])); }.
```

While this encoding loses explicit structure and results in a harder reasoning task, it is necessary in the context of the SV-Comp benchmark set, as other tools do not support explicit universal quantification in their input language. In contrast, our approach can handle arbitrarily quantified properties over unbounded data structures. We therefore directly formulated universally quantified properties, without using any program transformations. To improve the obtained set of benchmarks, we furthermore added challenging programs and additional functional properties to it.

We already presented an example `copy-positive` from ARRAYS in Figure 2.1, with a corresponding property (2.1). In Figure 9.1, two further examples from ARRAYS are presented, named `vector-addition` resp. `inc-by-1-harder`, with accompanying properties (9.1) resp. (9.2).

$$\forall pos^{\mathbb{I}}.\ \Big((0 \leq pos < length \wedge 0 \leq length) \rightarrow c(end, pos) \simeq a(pos) + b(pos)\Big) \tag{9.1}$$

$$\forall pos^{\mathbb{I}}.\ \Big((0 \leq pos < alength \wedge 0 \leq alength) \rightarrow a(end, pos) \simeq a(l_{12}, pos) + 1\Big) \tag{9.2}$$

The benchmark set ARRAYS is challenging. Not only do most examples require reasoning with theories and universal quantification, but many of them also require reasoning with existential or even alternating quantification. The need for such quantification is obvious for the 7 resp. 23 properties included in ARRAYS, as they already contain existential resp. alternating quantification *explicitly*. But we want to emphasize that also many other properties of ARRAYS require reasoning with existential or alternating quantification, even though the properties themselves do not contain such quantification explicitly. For

instance, consider the following two (correct) properties of the running example from Figure 2.1.

$$\forall bpos^{\mathbb{I}}.(0 \leq bpos < blength \rightarrow \exists apos^{\mathbb{I}}.a(apos) \simeq b(end, bpos))$$
$$\forall bpos^{\mathbb{I}}.(0 \leq bpos < blength \rightarrow 0 \leq b(end, bpos))$$

The former property contains alternating quantification, while the latter does not. But a proof of the latter property involves a subproof of the former property, and therefore also requires reasoning with alternating quantification.

Relational benchmarks. To compensate for the lack of general benchmarks for first-order hyperproperties, we collected a second benchmark set RELATIONAL[2], for evaluating our verification framework on relational properties. It contains 32 examples coming from security applications with a total of 64 properties. In a nutshell, RELATIONAL contains examples with (i) non-interference properties (Subsection 4.2.1), (ii) sensitivity properties (Subsection 4.2.1), and (iii) functional properties about comparators (see also [SD16]).

In Section 4.2, we already presented examples from RELATIONAL, which involve non-interference- and sensitivity-properties. As an example for comparators, consider the program `comp-lex-array` denoted in Figure 9.2. It implements a lexicographic comparator for two (potentially unbounded) arrays. The benchmark set RELATIONAL contains for this program several properties, which involve 2 and 3 traces and capture some form of symmetry, antisymmetry, and transitivity of the comparator. These properties are both interesting and challenging, as they require advanced loop splitting in combination with reasoning about multiple executions of the program.

9.2 The Tool Rapid

We implemented the trace-logic-based verification framework, which we introduced in Chapter 3 and generalized in Chapter 4, in the tool Rapid[3]. Our implementation consists of nearly 13,000 lines of C++ code.

Rapid takes as input a program $\mathtt{p}_0$ written in $\mathcal{W}$ (described in Section 2.1) and one or more properties expressed in trace logic $\mathcal{L}$ (introduced in Section 2.2). It then generates the axiomatic semantics $[\![\mathtt{p}_0]\!]$ (introduced in Section 2.4) of the program $\mathtt{p}_0$ and a set of trace lemmas $L_1, \ldots, L_m$ specific to the program $\mathtt{p}_0$ (as introduced in Subsection 3.2.1 and generalized in Subsection 4.1.3). Finally, it produces two sets of validity statements, with each validity statement written in trace logic $\mathcal{L}$ using extended SMT-LIB syntax (see Subsection 9.2.2), and being output into a separate file: On the one hand, to establish the correctness of the generated trace lemmas $L_1, \ldots, L_m$, Rapid generates for each trace lemma L_i a validity statement containing (i) the axioms corresponding

[2] Available at https://github.com/gleiss/rapid/examples/relational.
[3] Available at https://github.com/gleiss/rapid.

```
1  func main()
2  {
3    const Int[] a;
4    const Int[] b;
5    const Int length;
6    // return value: 20 encodes a<b,
7    // 10 encodes a=b, 30 encodes b<a
8    Int ret;
9
10   Int i = 0;
11   while(i < length && a[i] == b[i])
12   {
13     i = i + 1;
14   }
15
16   if (i < length)
17   {
18     if (a[i] < b[i])
19     {
20       ret = 20;
21     }
22     else
23     {
24       ret = 30;
25     }
26   }
27   else
28   {
29     ret = 10;
30   }
31 }
```

Figure 9.2: Example from RELATIONAL.

to relevant instances of the induction axioms schemes of difference logic[4], and (ii) the conjecture L_i. On the other hand, RAPID assumes the already-established correctness of the trace lemmas, and generates for each property of the input a validity statement, containing (i) $[\![\mathrm{p}_0]\!]$ and $L_1, \ldots, L_m$ as axioms, and (ii) the property as conjecture.

This way, to establish the partial correctness of a given property, we first prove the correctness of each trace lemma L_i, and then prove the partial correctness of the given property while assuming the correctness of all trace lemmas $L_1, \ldots, L_m$.

[4]For the At-Least-One-Iteration trace lemma and for the Simultaneous-Termination trace lemma, also the semantics of the program are added as axioms.

9.2.1 Optimizations to the Encoding

Similar to standard SSA-style compiler optimizations [App98, App04, WG12], Rapid heuristically simplifies the axiomatic semantics and the trace lemmas, by inlining (conditional) equalities occurring in the axiomatic semantics. In particular, while generating the axiomatic semantics and trace lemmas, Rapid keeps track of (i) the current (symbolic) program expression assigned to each integer variable and (ii) the last timepoint where each array-variable was assigned to. It then uses these cached values to heuristically simplify later program expressions. We will showcase the effect of this simplification using two examples.

9.1. Example Consider the program denoted in Figure 9.3a, and the property $x(end) < z(end)$. The original axiomatic semantics, as defined in Section 2.4, contains an equality for each of the variables x, y, and z, for each of the locations l_8, l_9, and *end*. In contrast, the *inlined* axiomatic semantics generated by Rapid are

$$\begin{aligned} &x(end) \simeq x(l_8) + 1 \ \wedge \\ &z(end) \simeq (x(l_8) + 1) + 2. \end{aligned}$$

The semantics defines the values of the program variables x and z at the end of the program, as these values occur in the property. Note that the definition of the value of the program variable z at the end of the program inlines the values of the program variables x and y at the timepoint of the last assignment. The variable y is not defined at all, although it is used to define the value of z in the last assignment, as its value is not used in the property. Similarly, the equalities defining the values of all program variables at the locations l_7 and l_9 as well as the equalities defining the values of y and z at location l_8 are inlined, as these values are irrelevant to the property. □

9.2. Example As a second example, consider the program denoted in Figure 9.3b, and the property $x(end) \geq 1$, for which the inlined axiomatic semantics is

$$\begin{aligned} &x(l_6) < 1 \rightarrow x(l_{14}, 0) \simeq 1 \ \wedge \\ &x(l_6) \not< 1 \rightarrow x(l_{14}, 0) \simeq x(l_6) \ \wedge \\ &x(end) \simeq x(l_{14}, 0). \end{aligned}$$

The former two conjuncts are included in the semantics, as the value of $x(l_{14})$ differs for the two branches of the if-then-else-statement at line 6. The latter conjunct is included, as x occurs in the property. Note that the values of x for all iterations different from the first iteration are not defined, as these values are irrelevant to the property. □

The above inlining of (conditional) equalities is computed by Rapid during the generation of the axiomatic semantics, using a simple and efficient analysis of the program tree.[5] It reduces the size of the resulting axiomatization considerably. In our experience,

[5] It would be more expensive to perform such inlining as a post-processing step on the formula level.

```
1   func main ()
2   {
3     Int x;
4     Int y;
5     Int z;
6
7     x = x + 1;
8     y = 2;
9     z = x + y;
10  }
11
```

(a) Multiple assignments.

```
1   func main ()
2   {
3     Int x;
4     Int y;
5
6     if (x < 1)
7     {
8       x = 0;
9     }
10    else
11    {
12      skip;
13    }
14    while (y > 0)
15    {
16      y = y - 1;
17    }
18  }
19
```

(b) Unassigned mutable variable in loop.

Figure 9.3: Examples showcasing SSA-style inlining.

the size of the encoding was reduced by 25 to 45 percent. The smaller axiomatization produced this way has two benefits. On the one hand, the performance of Vampire is increased, as many (conditional) equalities are omitted, which are irrelevant to the proof of the property. On the other hand, the manual inspection of proofs produced by Vampire is tremendously simplified, since many trivial rewriting-inferences are omitted.

As a second optimization, Rapid inlines all occurrences of the predicate *Reach* in the axiomatic semantics of the program. The main motivation for this optimization is that the proofs of the resulting encodings are easier to inspect manually.

9.2.2 Extended SMT-LIB Syntax

Rapid outputs validity statements in trace logic using SMT-LIB syntax [BFT17] (version 2.6). To preserve as much high-level structure as possible, it utilizes three custom extensions not covered by standard SMT-LIB syntax: First, it denotes the conjecture F of a given validity statement using the statement `(assert-not F)`, to enable goal-directed reasoning (see also Section 6.4 and [RR18]). Secondly, Rapid uses the custom statement `(declare-nat Nat 0 s p Sub)` to denote that the sort `Nat`, with zero symbol `0`, successor symbol `s`, predecessor symbol `p`, and ordering relation `Sub`, should be interpreted as theory $\mathbb{D}$ of difference logic over natural numbers. Thirdly, to distinguish lemma literals (which will be introduced in Section 9.3) from ordinary literals, Rapid uses the keyword `declare-lemma-predicate`, which should be understood

as an alias of `declare-function`, except that it additionally conveys the information that each occurrence of the declared symbol should be treated as a lemma literal.

9.3 A Custom Version of Vampire

In Section 5.3, we discussed how to tune Vampire for the trace logic domain using existing options. In this section, we tune Vampire further by extending it with custom techniques specific to the trace logic domain. This way, we provide Vampire with custom domain knowledge, which cannot be provided using the existing options. The techniques explained in this section are straightforward and easy to implement, and are at the same time important for efficient reasoning of Vampire on the trace logic domain.

9.3.1 Adapting the Literal Selection Heuristics for Trace Lemmas

We now show how to control in which way Vampire uses the trace lemmas from Subsection 3.2.1 to derive consequences during proof search. Intuitively, there are different possibilities to use lemmas: (i) try to prove the premises of a lemma L, and then, if this step succeeded, use the conclusion of L to derive further consequences, (ii) use the conclusion of a lemma L without establishing the premises first, and only aim to prove the premises of L, if the conclusion was used to derive an interesting subgoal, or (iii) interleave the proving of premises of a lemma L with the exploration of consequences of the conclusion of L. We refer to choice (i) as *forward reasoning*, choice (ii) as *backward reasoning*, and choice (iii) as *combined forward- and backward reasoning*. We argue that forward reasoning should be preferred for the trace lemmas of the trace logic domain: For most examples, the premises of only a few trace lemmas are fulfilled. Moreover, the conclusions of most trace lemmas are logically strong facts, and can therefore be used to derive many consequences, which are all unusable, if the premises of the trace lemma do not hold.

We now discuss how to force Vampire to use forward reasoning for trace lemmas as often as possible (without giving up completeness): Consider a trace lemma L of the form

$$(P_1 \wedge \cdots \wedge P_k) \rightarrow \mathit{Conclusion},$$

where $P_1, \ldots, P_k$ are arbitrary literals, and $\mathit{Conclusion}$ denotes an arbitrary conclusion literal.[6] For any trace lemma L, we first introduce a naming literal Prem_L denoting that the premises of L hold, and replace L by the two formulas

$$(P_1 \wedge \cdots \wedge P_k) \rightarrow \mathit{Prem}_L \tag{9.3}$$

$$\mathit{Prem}_L \rightarrow \mathit{Conclusion} \tag{9.4}$$

[6] To simplify the presentation, we only discuss the case where the premises and the conclusion of the trace lemma consist of a single literal. The general case, where the premises and conclusions are arbitrary formulas, can be handled in a similar way, but requires a non-trivial CNF-transformation.

We refer to all naming literals introduced in this way as *lemma literals*. Clausifying the formulas (9.3) and (9.4) will produce clauses of the form

$$\neg P_1 \vee \cdots \vee \neg P_k \vee Prem_L \tag{9.5}$$

$$\neg Prem_L \vee Conclusion \tag{9.6}$$

Our goal is to force Vampire to (i) prove $P_1, \ldots, P_k$, so that it can conclude $Prem_L$ using (9.5), and (ii) only afterwards resolve $Prem_L$ with $\neg Prem_L$ to conclude *Conclusion* using (9.6). The order in which literals are proven and combined together in Vampire is controlled by the literal selection heuristics. The requirement (i) of our goal is translated to the property of the literal selection function that a positive lemma literal should never be the only selected literal, if another literal exists. The requirement (ii) of our goal is translated to the property that we should always select a negative lemma literal (and no other literals), if such a literal exists in a clause.

While manually inspecting proof attempts on the trace logic domain, we observed that none of the literal selection heuristics discussed in Section 5.2 could realize the exploration strategy discussed above.[7]

To solve this problem, we adapt each literal selection heuristics *ls* discussed in Section 5.2 as follows: Before we select any literal of a clause C, we check whether a negative lemma literal exists in C. If this is the case, we select the negative lemma literal and no other literal. Note that this change preserves the potential well-behavedness of *ls*, as we are always allowed to select only a single negative literal. If no negative lemma literal exists, we proceed with selecting literals according to *ls*. Finally, we check whether only positive lemma literals have been selected, although at least one non-lemma-literal exists in C. If this is the case, we select an additional non-lemma literal (using *ls*). This step also preserves the well-behavedness of literal selection heuristics, as we can always select additional literals without compromising the well-behavedness. Even though we select in this step more literals than necessary to ensure well-behavedness, this modification improves the performance of Vampire on the trace logic domain in our experience. As a side effect, we observed that the discussed changes to the literal selection heuristics dramatically improve the locality[8] of many proof attempts in the trace logic domain, and therefore considerably simplify their manual inspection.

9.3.2 A Custom Deletion Rule for Inequalities of Difference Logic

As discussed in Section 5.2, we extend the theory of term algebras with an ordering relation, to enable reasoning modulo difference logic. Unfortunately, the theory axioms

[7] First, if a premise of a trace lemma is a disequality, it would be clausified into a positive equality, which would not be selected earlier than $Prem_L$ by any of the complete literal selection heuristics `10` and `11`. Secondly, we observed that the literal *Conclusion* is often both bigger in the ordering and has higher weight than $\neg Prem_L$, so `10` and `1010` would select *Conclusion* instead of $\neg Prem_L$. Thirdly, we observed that resolving $Prem_L$ and $\neg Prem_L$ often produces only a small number of consequences, so `11` and `1011` would often select these literals.

[8] Intuitively, the locality of a proof corresponds to the average number of inferences a literal participates in before it is selected (with a small number being preferable).

added in this way are problematic, as they generate a lot of useless consequences. During a manual inspection of proof attempts for the trace logic domain, we observed that many clauses with terms of the form $\texttt{succ}(\texttt{0})$ or $\texttt{succ}(\texttt{succ}(t))$ (for some subterm t) were generated. We further observed that such terms are not needed in the examples from the trace logic domain we are interested in. We therefore introduce a deletion inference rule called *custom-successor-deletion*, which deletes any clause containing a term of the form $\texttt{succ}(\texttt{0})$ or of the form $\texttt{succ}(\texttt{succ}(t))$. We want to note that this deletion rule should be interpreted as a lightweight technique, which handles the blowup of the search space introduced by inequality reasoning for difference logic and improves the performance of Vampire on the trace logic domain. We acknowledge that our handling of difference logic could potentially be vastly improved by a more principled approach. However, such an approach would require a huge effort and is out of the scope of this thesis.

9.3.3 Implementation

As already described in Chapter 6 and Chapter 7, we implemented the layered clause selection framework resp. the subsumption demodulation simplification rule in Vampire. Starting with release 4.5, our implementation is part of the official version of Vampire.

For our experiments, we extended Vampire 4.5[9] with a custom implementation[10] to realize the ideas described in Subsection 9.3.1 and Subsection 9.3.2. The custom implementation adds a native (background) theory of difference logic to Vampire, including (i) parsing of statements `(declare-nat Nat 0 s p Sub)`, (ii) internally added theory axioms for the ordering relation of difference logic (presented in Figure 5.2), and (iii) the deletion inference custom-successor deletion (controlled by option `-csd`, with possible values `off` and `on`, and default value `off`). Moreover, it adds support for lemma literals, including (i) parsing of the keyword `declare-lemma-predicate`, and (ii) adapted literal selection heuristics with custom handling for lemma literals (controlled by option `-lls`, with possible values `off` and `on`, and default value `off`). Finally, the extended implementation fine-tunes the existing internal theory axioms for integers, by adding the theory axiom $x \not\simeq x + 1$. While this axiom does not add additional strength logically, it enables additional simplifications in combination with subsumption resolution, as any clause of the form $C \vee t \simeq t + 1$ will be simplified to C using subsumption resolution with the side premise $x \not\simeq x + 1$.

9.3.4 Configurations

We now describe the (portfolio) configurations of Vampire used for the experimental evaluation.

Base configuration. Recall the discussion from Section 5.3 on how to configure Vampire for the trace logic domain. We realized these ideas in a portfolio configuration of

[9] Commit 57a6f78cc2bc179e480c3d47d09fbfc1ec8d1f23.
[10] Available at https://github.com/vprover/vampire/tree/gleiss-rapid.

two strategies, which we will refer to as conf, as follows. Starting from Vampire's default strategy, we set the options `--input_syntax smtlib2`, `--newcnf on`, `-t 60`, `-av off`, `-bs on`, `-bsr on`, `-urr on`. `-nwc 2`. Additionally, we vary the option `-s` between the two values `10` and `1010`. The configuration conf is interesting, as it shows how well Vampire does perform on the trace logic domain, if we tune it with already existing options, but do not use any technique implemented in Vampire as part of this thesis.

Advanced configuration. The configuration adv refines the configuration conf using all the reasoning techniques developed in this thesis, and represents the most advanced configuration of Vampire for reasoning in the trace logic domain. It differs from the configuration conf as follows.

First, it uses the layered clause selection framework introduced in Chapter 6. In particular, it uses the clause selection heuristics

$$\begin{aligned} &\mathit{mono\text{-}split}(\mathit{dist}^{8}_{th}, \_, \_ \\ &\quad \mathit{mono\text{-}split}(\mathit{dist}_{Horn}, \_, \_ \\ &\quad\quad \mathit{mono\text{-}split}(\mathit{dist}_{SInE}, \_, \_ \\ &\quad\quad\quad \mathit{aw}(1:1)))), \end{aligned}$$

consisting of three nested monotone split heuristics with features dist^{8}_{th}, dist_{Horn} and dist_{SInE}, where the cutoffs and ratios of the split heuristics are varied in the portfolio. For the split heuristics with feature dist^{8}_{th}, we use a variant which combines cutoffs (0,8) and ratio 20:10:1, and a variant which combines cutoffs (0,8,16,24) with ratio 20:10:10:10:1. For the split heuristics with feature dist_{Horn}, we fix the cutoffs (1,2) and vary the ratio between 1:1:1, 5:5:1, 10,10,1, and 20:20:1. Furthermore, we add a variant, where the split heuristics with feature dist_{Horn} is not used. For the split heuristics with feature dist_{SInE}, we fix the cutoff (0) and vary the ratio between 1:1 and 30:1. Additionally, we also add a variant, where the split heuristics with feature dist_{SInE} is not used. Summing up, we obtain 2 variants for the split heuristics with feature dist^{8}_{th}, 5 variants for the split heuristics with feature dist_{Horn}, and 3 variants for the split heuristics with feature dist_{SInE}, yielding 30 overall variants of the clause selection heuristics.[11]

Secondly, the configuration uses the simplifying inference rule subsumption demodulation introduced in Chapter 7, by setting for each strategy the options `-fsd on` and `-bsd on`.

Thirdly, it uses the simple techniques described in Section 9.3, by setting for each strategy the options `-lls on` and `-csd on`.

[11] The necessary options to enable these variants in Vampire are presented in Section 6.6. For instance, one of these variants could be enabled using `-thsq on -thsqd 8 -thsqc 0,8 -thsqr 20,10,1 -thsql on -plsq on -plsqc 1,2 -plsqr 1,1,1 -plsql on -slsq on -slsqc 0 -slsqr 1,1 -slsql on`.

Additional configurations. The configurations NO-MSQ and NO-SD are obtained from the configuration ADV by turning off layered clause selection resp. subsumption demodulation. These configurations are interesting for understanding how much the performance of the configuration ADV depends on layered clause selection resp. subsumption demodulation.

9.4 Experimental Evaluation

We now present an experimental evaluation of our tool RAPID (Section 9.2) and our custom version of VAMPIRE (Section 9.3). All experiments were run on an Intel Core i5 3.1Ghz machine with 16 GB of RAM.

For each example and each property from the benchmark sets discussed in Section 9.1, we first use RAPID to generate a validity problem encoding the partial correctness of the property with respect to the given program (as discussed in Section 9.2), and then use VAMPIRE to prove that validity problem. As discussed in Section 9.2, for each example, RAPID additionally generates a set of validity problems encoding the correctness of the trace lemmas used to establish the partial correctness of the properties of the given example. We successfully prove each of these validity problems in less than 1 second by separately invoking VAMPIRE in the configuration ADV.

9.4.1 Experiment 1 - Overall Performance

As a first experiment, we evaluated the configurations CONF and ADV on the benchmark sets ARRAYS and RELATIONAL. The results are presented in Table 9.1 and Table 9.2.

First, we are interested in how many properties we can verify using the configuration ADV, which utilizes all the techniques developed in this thesis. We can see that ADV proves 78 out of 103 examples from the benchmark set ARRAYS and that it proves 57 out of 64 examples from the benchmark set RELATIONAL. As both sets of benchmarks contain many challenging properties, which require reasoning with quantifier-alternations and/or reasoning about multiple traces, we argue that our approach works quite well.

Secondly, we are interested in how many properties we could have solved using the configuration CONF, which is obtained from tuning VAMPIRE to the trace logic domain using only user options and in particular without using any of the reasoning techniques introduced in this thesis. We can see that configuration CONF only proves 18 out of 103 examples from the benchmark set ARRAYS and only proves 32 out of 64 examples from the benchmark set RELATIONAL. In particular, the configuration ADV improves over CONF by 60 examples on ARRAYS and by 25 examples on RELATIONAL, which suggests that the reasoning techniques developed in this thesis are essential to enable efficient reasoning with VAMPIRE in the trace logic domain.

Table 9.1: Results of Experiment 1 on the benchmark set ARRAYS.

Benchmark	CONF	ADV	Benchmark	CONF	ADV
atleast-one-iteration-0	✓	✓	inc-by-one-harder-0	-	✓
atleast-one-iteration-1	✓	✓	inc-by-one-harder-1	-	✓
both-or-none-0	-	✓	indexn-is-arraylength-0	✓	✓
check-equal-set-flag-0	✓	✓	indexn-is-arraylength-1	✓	✓
check-equal-set-flag-1	-	✓	init-0	-	✓
collect-indices-0	-	✓	init-conditionally-0	-	✓
collect-indices-1	-	✓	init-conditionally-1	-	✓
collect-indices-2	-	✓	init-non-const-0	-	✓
collect-indices-3	-	-	init-non-const-1	-	✓
copy-0	-	✓	init-non-const-2	-	✓
copy-absolute-0	-	✓	init-non-const-3	-	✓
copy-absolute-1	-	✓	init-non-const-easy-0	-	✓
copy-nonzero-0	-	✓	init-non-const-easy-1	-	✓
copy-nonzero-1	-	✓	init-non-const-easy-2	-	✓
copy-partial-0	-	✓	init-non-const-easy-3	-	✓
copy-positive-0	-	✓	init-partial-0	-	✓
copy-positive-1	-	✓	init-prev-plus-one-0	-	✓
copy-two-indices-0	-	✓	init-prev-plus-one-1	-	✓
find1-0	✓	✓	init-prev-plus-one-alt-0	-	✓
find1-1	-	✓	init-prev-plus-one-alt-1	-	✓
find1-2	✓	✓	max-prop-0	-	✓
find1-3	✓	✓	max-prop-1	-	✓
find1-4	-	✓	merge-interleave-0	-	-
find2-0	✓	✓	merge-interleave-1	-	-
find2-1	✓	✓	merge-interleave-2	-	-
find2-2	✓	✓	min-prop-0	-	✓
find2-3	✓	✓	min-prop-1	-	✓
find2-4	✓	✓	partition-0	-	✓
find-max-0	-	✓	partition-1	-	✓
find-max-1	-	-	partition-2	-	✓
find-max-2	-	✓	partition-3	-	✓
find-max-from-second-0	-	-	partition-4	-	-
find-max-from-second-1	-	-	partition-5	-	✓
find-max-local-0	-	-	partition-6	-	-
find-max-local-1	-	-	partition-harder-0	-	✓
find-max-local-2	-	-	partition-harder-1	-	✓
find-max-up-to-0	-	-	partition-harder-2	-	-
find-max-up-to-1	-	-	partition-harder-3	-	-
find-max-up-to-2	-	-	partition-harder-4	-	-
find-min-0	-	✓	push-back-0	-	✓
find-min-1	-	-	reverse-0	-	✓
find-min-2	-	✓	set-to-one-0	✓	✓
find-min-local-0	-	-	str-cpy-0	-	✓
find-min-local-1	-	-	str-cpy-1	-	✓
find-min-local-2	-	-	str-cpy-2	✓	✓
find-min-up-to-0	-	-	str-cpy-3	✓	✓
find-min-up-to-1	-	-	str-len-0	✓	✓
find-min-up-to-2	-	-	swap-0	-	✓
find-sentinel-0	✓	✓	swap-1	-	✓
in-place-max-0	-	✓	vector-addition-0	-	✓
inc-by-one-0	-	✓	vector-subtraction-0	-	✓
inc-by-one-1	-	✓	**Total**	**18**	**78**

Table 9.2: Results of Experiment 1 on the benchmark set RELATIONAL.

Benchmark	CONF	ADV	Benchmark	CONF	ADV
1-hw-equal-arrays-0	-	✓	8-sens-explicit-swap-0	-	✓
1-hw-equal-arrays-harder-0	-	-	8-sens-explicit-swap-1	-	✓
1-ni-assign-to-high-0	✓	✓	8-sens-explicit-swap-2	-	✓
1-ni-equal-output-0	-	✓	comp-lex-array-1-trace-0	✓	✓
1-sens-equal-sums-0	-	✓	comp-lex-array-1-trace-1	✓	✓
10-ni-rsa-exponentiation-0	✓	✓	comp-lex-array-1-trace-2	-	✓
10-sens-equal-k-0	-	✓	comp-lex-array-1-trace-3	✓	✓
11-sens-equal-k-twice-0	-	✓	comp-lex-array-2-traces-0	-	✓
12-sens-diff-up-to-forall-k-0	-	✓	comp-lex-array-2-traces-1	-	✓
2-hw-last-position-swapped-0	-	✓	comp-lex-array-3-traces-0	-	✓
2-hw-last-position-swapped-harder-0	-	-	comp-lex-array-3-traces-1	-	-
2-ni-branch-on-high-0	✓	✓	comp-lex-array-3-traces-2	-	-
2-ni-branch-on-high-1	-	✓	comp-lex-pair-1-trace-0	✓	✓
2-ni-branch-on-high-twice-0	✓	✓	comp-lex-pair-1-trace-1	✓	✓
2-ni-branch-on-high-twice-1	-	✓	comp-lex-pair-1-trace-2	✓	✓
2-sens-equal-sums-two-arrays-0	-	✓	comp-lex-pair-1-trace-3	✓	✓
3-hw-swap-and-two-arrays-0	-	✓	comp-lex-pair-2-traces-0	✓	✓
3-hw-swap-and-two-arrays-harder-0	-	-	comp-lex-pair-2-traces-1	✓	✓
3-ni-high-guard-equal-branches-0	✓	✓	comp-lex-pair-3-traces-0	✓	✓
3-sens-abs-diff-up-to-k-0	-	✓	comp-lex-pair-3-traces-1	✓	✓
4-hw-swap-in-array-full-0	-	-	comp-lex-pair-3-traces-2	✓	✓
4-hw-swap-in-array-lemma-0	-	-	comp-lex-pair-3-traces-3	-	✓
4-sens-abs-diff-up-to-k-two-arrays-0	-	✓	comp-lex-single-1-trace-0	✓	✓
5-ni-temp-impl-flow-0	✓	✓	comp-lex-single-1-trace-1	✓	✓
5-ni-temp-impl-flow-1	✓	✓	comp-lex-single-1-trace-2	✓	✓
5-ni-temp-impl-flow-2	✓	✓	comp-lex-single-1-trace-3	✓	✓
5-sens-two-arrays-equal-k-0	-	✓	comp-lex-single-2-traces-0	✓	✓
6-ni-branch-assign-equal-val-0	-	✓	comp-lex-single-2-traces-1	✓	✓
6-sens-diff-up-to-explicit-k-0	-	✓	comp-lex-single-3-traces-0	✓	✓
7-ni-explicit-flow-0	✓	✓	comp-lex-single-3-traces-1	✓	✓
7-sens-diff-up-to-explicit-k-sum-0	-	✓	comp-lex-single-3-traces-2	✓	✓
8-ni-explicit-flow-while-0	✓	✓	comp-lex-single-3-traces-3	✓	✓
			Total	**32**	**57**

9.4.2 Experiment 2 - Contribution of New Techniques

As a second experiment, we investigated the separate contribution of layered clause selection and subsumption demodulation - the two main reasoning techniques developed in this thesis - to the efficient performance of VAMPIRE's configuration ADV on the trace logic domain. To this end, we evaluated the configurations NO-MSQ and NO-SD, obtained from the best configuration ADV by turning off layered clause selection resp. by turning off subsumption demodulation, on the benchmark sets ARRAYS and RELATIONAL, and compared them with the configuration ADV. Without layered clause selection, we prove only 19 out of 103 examples from ARRAYS and only 33 out of 64 examples from RELATIONAL. We therefore prove only slightly more examples than we did using the configuration CONF, suggesting that layered clause selection is a critical ingredient to the performance of ADV. Secondly, without subsumption demodulation, we prove 76 out

Table 9.3: Results of Experiment 2 on the benchmark set ARRAYS.

Benchmark	ADV	NO-MSQ	NO-SD	Benchmark	ADV	NO-MSQ	NO-SD
atleast-one-iteration-0	✓	✓	✓	inc-by-one-harder-0	✓	-	✓
atleast-one-iteration-1	✓	✓	✓	inc-by-one-harder-1	✓	-	✓
both-or-none-0	✓	-	✓	indexn-is-arraylength-0	✓	✓	✓
check-equal-set-flag-0	✓	✓	✓	indexn-is-arraylength-1	✓	✓	✓
check-equal-set-flag-1	✓	-	✓	init-0	✓	-	✓
collect-indices-0	✓	-	✓	init-conditionally-0	✓	-	✓
collect-indices-1	✓	-	✓	init-conditionally-1	✓	-	✓
collect-indices-2	✓	-	-	init-non-const-0	✓	-	✓
collect-indices-3	-	-	-	init-non-const-1	✓	-	✓
copy-0	✓	-	✓	init-non-const-2	✓	-	✓
copy-absolute-0	✓	-	✓	init-non-const-3	✓	-	✓
copy-absolute-1	✓	-	✓	init-non-const-easy-0	✓	-	✓
copy-nonzero-0	✓	-	✓	init-non-const-easy-1	✓	-	✓
copy-nonzero-1	✓	-	✓	init-non-const-easy-2	✓	-	✓
copy-partial-0	✓	-	✓	init-non-const-easy-3	✓	-	✓
copy-positive-0	✓	-	✓	init-partial-0	✓	-	✓
copy-positive-1	✓	-	✓	init-prev-plus-one-0	✓	-	✓
copy-two-indices-0	✓	-	✓	init-prev-plus-one-1	✓	-	✓
find1-0	✓	✓	✓	init-prev-plus-one-alt-0	✓	-	✓
find1-1	✓	✓	✓	init-prev-plus-one-alt-1	✓	-	✓
find1-2	✓	✓	✓	max-prop-0	✓	-	✓
find1-3	✓	✓	✓	max-prop-1	✓	-	✓
find1-4	✓	-	-	merge-interleave-0	-	-	-
find2-0	✓	✓	✓	merge-interleave-1	-	-	-
find2-1	✓	✓	✓	merge-interleave-2	-	-	-
find2-2	✓	✓	✓	min-prop-0	✓	-	✓
find2-3	✓	✓	✓	min-prop-1	✓	-	✓
find2-4	✓	✓	✓	partition-0	✓	-	✓
find-max-0	✓	-	✓	partition-1	✓	-	✓
find-max-1	-	-	-	partition-2	✓	-	✓
find-max-2	✓	-	✓	partition-3	✓	-	✓
find-max-from-second-0	-	-	-	partition-4	-	-	-
find-max-from-second-1	-	-	-	partition-5	✓	-	✓
find-max-local-0	-	-	-	partition-6	-	-	-
find-max-local-1	-	-	-	partition-harder-0	✓	-	✓
find-max-local-2	-	-	-	partition-harder-1	✓	-	✓
find-max-up-to-0	-	-	-	partition-harder-2	-	-	-
find-max-up-to-1	-	-	-	partition-harder-3	-	-	-
find-max-up-to-2	-	-	-	partition-harder-4	-	-	-
find-min-0	✓	-	✓	push-back-0	✓	-	✓
find-min-1	-	-	-	reverse-0	✓	-	✓
find-min-2	✓	-	✓	set-to-one-0	✓	✓	✓
find-min-local-0	-	-	-	str-cpy-0	✓	-	✓
find-min-local-1	-	-	-	str-cpy-1	✓	-	✓
find-min-local-2	-	-	-	str-cpy-2	✓	✓	✓
find-min-up-to-0	-	-	-	str-cpy-3	✓	✓	✓
find-min-up-to-1	-	-	-	str-len-0	✓	✓	✓
find-min-up-to-2	-	-	-	swap-0	✓	-	✓
find-sentinel-0	✓	✓	✓	swap-1	✓	-	✓
in-place-max-0	✓	-	✓	vector-addition-0	✓	-	✓
inc-by-one-0	✓	-	✓	vector-subtraction-0	✓	-	✓
inc-by-one-1	✓	-	✓	**Total**	**78**	**19**	**76**

Table 9.4: Results of Experiment 2 on the benchmark set RELATIONAL.

Benchmark	ADV	NO-MSQ	NO-SD	Benchmark	ADV	NO-MSQ	NO-SD
1-hw-equal-arrays-0	✓	-	✓	8-sens-explicit-swap-0	✓	-	-
1-hw-equal-arrays-harder-0	-	-	-	8-sens-explicit-swap-1	✓	-	-
1-ni-assign-to-high-0	✓	✓	✓	8-sens-explicit-swap-2	✓	-	-
1-ni-equal-output-0	✓	-	-	comp-lex-array-1-trace-0	✓	✓	✓
1-sens-equal-sums-0	✓	-	✓	comp-lex-array-1-trace-1	✓	✓	✓
10-ni-rsa-exponentiation-0	✓	✓	✓	comp-lex-array-1-trace-2	✓	-	✓
10-sens-equal-k-0	✓	-	-	comp-lex-array-1-trace-3	✓	-	✓
11-sens-equal-k-twice-0	✓	-	-	comp-lex-array-2-traces-0	✓	-	✓
12-sens-diff-up-to-forall-k-0	✓	-	✓	comp-lex-array-2-traces-1	✓	-	-
2-hw-last-position-swapped-0	✓	-	-	comp-lex-array-3-traces-0	✓	-	-
2-hw-last-position-swapped-harder-0	-	-	-	comp-lex-array-3-traces-1	-	-	-
2-ni-branch-on-high-0	✓	✓	✓	comp-lex-array-3-traces-2	-	-	-
2-ni-branch-on-high-1	✓	-	✓	comp-lex-pair-1-trace-0	✓	✓	✓
2-ni-branch-on-high-twice-0	✓	✓	✓	comp-lex-pair-1-trace-1	✓	✓	✓
2-ni-branch-on-high-twice-1	✓	-	✓	comp-lex-pair-1-trace-2	✓	✓	✓
2-sens-equal-sums-two-arrays-0	✓	-	-	comp-lex-pair-1-trace-3	✓	✓	✓
3-hw-swap-and-two-arrays-0	✓	-	-	comp-lex-pair-2-traces-0	✓	✓	✓
3-hw-swap-and-two-arrays-harder-0	-	-	-	comp-lex-pair-2-traces-1	✓	✓	✓
3-ni-high-guard-equal-branches-0	✓	✓	✓	comp-lex-pair-3-traces-0	✓	✓	✓
3-sens-abs-diff-up-to-k-0	✓	-	✓	comp-lex-pair-3-traces-1	✓	✓	✓
4-hw-swap-in-array-full-0	-	-	-	comp-lex-pair-3-traces-2	✓	✓	✓
4-hw-swap-in-array-lemma-0	-	-	-	comp-lex-pair-3-traces-3	✓	✓	✓
4-sens-abs-diff-up-to-k-two-arrays-0	✓	-	-	comp-lex-single-1-trace-0	✓	✓	✓
5-ni-temp-impl-flow-0	✓	✓	✓	comp-lex-single-1-trace-1	✓	✓	✓
5-ni-temp-impl-flow-1	✓	✓	✓	comp-lex-single-1-trace-2	✓	✓	✓
5-ni-temp-impl-flow-2	✓	✓	✓	comp-lex-single-1-trace-3	✓	✓	✓
5-sens-two-arrays-equal-k-0	✓	-	-	comp-lex-single-2-traces-0	✓	✓	✓
6-ni-branch-assign-equal-val-0	✓	✓	✓	comp-lex-single-2-traces-1	✓	✓	✓
6-sens-diff-up-to-explicit-k-0	✓	-	-	comp-lex-single-3-traces-0	✓	✓	✓
7-ni-explicit-flow-0	✓	✓	✓	comp-lex-single-3-traces-1	✓	✓	✓
7-sens-diff-up-to-explicit-k-sum-0	✓	-	-	comp-lex-single-3-traces-2	✓	✓	✓
8-ni-explicit-flow-while-0	✓	✓	✓	comp-lex-single-3-traces-3	✓	✓	✓
				Total	**57**	**33**	**42**

of 103 examples from ARRAYS and 42 out of 64 examples from RELATIONAL. While this suggests that subsumption demodulation does not contribute much to the performance of ADV on the benchmark set ARRAYS, it also suggests that subsumption demodulation considerably improves the performance of ADV on the benchmark set RELATIONAL.

CHAPTER 10

Conclusion and Future Work

10.1 Conclusion

We presented a new approach to software verification, which leverages superposition-based theorem proving.

In the first part of the thesis, we developed a new verification framework for establishing the partial correctness of software programs. As main feature, we introduced trace logic, a new instance of first-order logic modulo difference logic and integer arithmetic, and formulated an axiomatic semantics of software programs in it. Trace logic keeps loop iterations explicit, enabling us (i) to describe each timepoint in the execution of a program uniquely, and (ii) to formulate instances of the generalized induction axiom scheme directly as logical formulas. To understand the theoretical foundations of the introduced axiomatic semantics, we furthermore established its soundness with respect to small-step operational semantics and its completeness with respect to Hoare logic. We then captured the partial correctness of a program property as a validity statement in trace logic, without using any intermediate program logic. We want to note that the trade-off of our semantics between the amount of quantification and the expressiveness of the logical language differs from the trade-off used by existing SMT-based approaches: Our semantics does not abstract away loop iterations and thereby involves more quantification, but in return is able to express generalized induction axioms, which can be used to perform advanced loop splitting.

Building on top of trace logic, we introduced a verification framework, which focuses on establishing the partial correctness of properties for programs involving (unbounded) arrays and loops. Since we defined the axiomatic semantics of trace logic using the semantics of standard first-order logic, we were immediately able to apply an arbitrary off-the-shelf first-order theorem prover to reason about the resulting validity statements. To overcome the problem that current state-of-the-art theorem provers do not provide

inductive reasoning that is sophisticated enough to automatically identify instances of the generalized induction axiom scheme necessary for the trace logic domain, we presented relevant instances of the generalized induction axiom scheme, which apply to a wide range of programs, and described how we instrumented our framework, to automatically generate these induction axioms for any given program and provide them to the used theorem prover as additional input axioms.

To showcase the flexibility of our approach, we then generalized trace logic and the trace-logic-based verification framework to support multiple traces and hyperproperties coming from security applications. We think that the presented generalization is compelling, as it is much simpler than existing approaches, which is possible due to trace logic's explicit notion of loop iterations and the direct formalization in standard first-order logic.

In the second part of the thesis, we turned our focus to the efficient application of the state-of-the-art superposition-based theorem prover Vampire to the trace logic domain, by using Vampire as reasoning engine inside the presented verification framework. We chose Vampire, since we conjectured that the efficient handling of quantification in superposition-based theorem proving is sufficient to enable efficient reasoning in the trace logic domain. Furthermore, we think that our work makes a significant step towards understanding the potential of superposition-based software verification. In contrast to SMT-based software verification, the boundaries of superposition-based software verification are not understood well, as principled applications of superposition-based theorem proving to software verification have not seen a lot of attention in the literature so far[1], although software verification has been reported as a main motivation for many recently developed techniques in superposition-based theorem proving [KRV17, Vor14, GKKV14, KKRV16, KKV18, KV09b, BLDM11, RV19]. Moreover, we argue that our combination of superposition-based theorem proving with trace logic (instead of an existing formalization of program correctness) reveals new advantages of superposition-based software verification, as Vampire's efficient reasoning with quantification in combination with trace logic's explicit notion of loop iterations enables advanced automated loop splitting.

After recalling the main ideas of superposition-based theorem proving, we described the techniques used in Vampire to realize these ideas. We then highlighted the places where Vampire offers multiple techniques from which a user can choose from, and argued which of these techniques to choose for applying Vampire to the trace logic domain.

Then, we presented two reasoning techniques that are designed to overcome inefficiencies of superposition-based theorem proving in the trace logic domain. On the one hand, we introduced layered clause selection heuristics to control the high-level proof search in saturation-based theorem proving. These heuristics guide the exploration of the search space towards proofs, which (i) are related to the conjecture we want to prove, and (ii)

[1] To the best of our knowledge, no verification framework has been described in the literature, which (i) targets a while-like language, (ii) uses a superposition-based theorem prover as backend, and (iii) formulates an axiomatic semantics in first-order logic modulo theories, which is specifically designed to enable efficient superposition-based theorem proving.

only contain reasonable amounts of theory reasoning and case distinctions. Our experiments show that the layered clause selection heuristics improve Vampire's efficiency on the trace logic domain drastically. Furthermore, our experience strongly suggests that they should also be essential in many other domains originating from industrial verification applications. On the other hand, we presented a simplification inference rule called subsumption demodulation, which enables efficient reasoning with conditional equalities. Our experiments show that subsumption demodulation is important in the trace logic domain, in particular to establish the partial correctness of hyperproperties. Moreover, we think that it is also useful in other application domains, as it both improves the stability of proof search and eases the human inspection of saturation attempts considerably.

Afterwards, we described the tool SatVis, which interactively visualizes saturation attempts of Vampire and therefore enables us to efficiently analyze such saturation attempts. SatVis was invaluable during the development of our trace-logic-based verification approach, as it allowed us to efficiently understand the effect of different choices for encodings and proof search techniques on the performance of Vampire in the trace logic domain.

Finally, we evaluated our work on a large set of challenging benchmarks. Our results show that the trace-logic-based verification framework in combination with our customized version of Vampire can automatically establish the partial correctness of many challenging examples. They suggest that our approach could be an interesting alternative to SMT-based approaches to software verification, in particular for programs involving (unbounded) arrays and loops. If we take a closer look at the results, we can also see that the reasoning techniques introduced in this thesis are crucial to the performance of Vampire on the trace logic domain. In particular, the results show that tuning Vampire only with existing options would not have been sufficient. This demonstrates that heavily customizing a superposition-based theorem prover to the software verification domain is rewarding, and in particular suggests that conclusions about the potential performance of a superposition-based prover should only be drawn after such customization has been performed.

10.2 Future Work

There are several directions to improve trace-logic-based software verification in the future, including (i) extensions of trace logic to support additional program constructs, (ii) new reasoning techniques for Vampire, and (iii) a more efficient development process for applications of Vampire to given domains. In the remainder of this section, we describe four concrete directions for future research.

10.2.1 Trace logic - Support for Function Calls

Trace logic, as described in this thesis, focuses on the analysis of programs consisting of a single function. In the future, it would be interesting to generalize trace logic to

```
1  func main()
2  {
3    const Int[] a;
4    Int[] b;
5    const Int len;
6
7    Int i = 0;
8    while(i < len)
9    {
10     b[i] = positive(a[i]);
11     i = i + 1;
12   }
13 }
```

```
1    func positive(Int v)
2    {
3      if (v >= 0)
4      {
5        return v;
6      }
7      else
8      {
9        return 0;
10     }
11   }
```

Figure 10.1: Program with multiple functions.

support multiple functions and function calls, to enable an interprocedural analysis. An example program requiring such an analysis is denoted in Figure 10.1. It contains a function `positive`, which is repeatedly called by the function `main`.

A standard solution for extending an existing intraprocedural analysis to an interprocedural analysis is to distinguish different invocations of a function by keeping track of the calling context which led to the invocation of the function [SP+78]. To realize such a solution, one has to (i) come up with a formalization of calling contexts, (ii) extend variable values so that they depend on the calling context of the function call, and (iii) define the semantics of calling a function and returning from it.

We conjecture that trace logic could offer an interesting solution to point (i). Existing approaches usually model the calling context as a call stack, that is, as a *list*[2] *of locations*. With such an approach, any invocation of the function `positive` from line 10 of the function `main` would be assigned the context $[main_{l_{10}}]$. With trace logic, we could in contrast choose a more expressive formalization, where we model the context as a *list of timepoints*. We would then capture the context for the invocation of the function `positive` from line 10 of the function `main` for a given iteration it_0 as $[main_{l_{10}}(it_0)]$. In particular, our formalization would *preserve the iteration* of the loop at line 8 of `main`, in which `positive` was called.

Taking a step back, we can see that function invocations can be uniquely described using a combination of locations and loop iterations. Most verification approaches already abstract away from loop iterations in the intraprocedural setting, and therefore are forced to model calling contexts in the interprocedural setting in an imprecise way without referring to loop iterations. In contrast, trace logic describes each timepoint in the execution of a program uniquely in the intraprocedural setting, and can therefore model

[2] Lists can be modeled in first-order logic using a term algebra with constructors `nil` and `cons`. Vampire already supports efficient reasoning with such term algebras [KRV17].

calling contexts as lists of timepoints in the interprocedural setting. As a result, it can refer to each function invocation uniquely.

10.2.2 Trace logic - Support for Unstructured Control-Flow Constructions

Trace logic supports standard structured control-flow-statements, including arbitrarily nested if-then-else- and while-statements. Extending our approach to other structured control-flow-statements, such as if-statements (without an else-branch), C-like for-loops, and case distinctions, would require some engineering effort, but is otherwise straightforward. More interestingly, one could generalize trace logic even further to support *unstructured* control-flow statements, including `break`-, `continue`-, restricted forms of `goto`-, and early-`return`-statements (we call a return-statement *early*, if it is not located at the end of the program). Such statements let the execution of the program jump to some statement[3] (given either implicitly or explicitly in the form of a label), which is potentially far away in the program tree from the unstructured control-flow statement.

10.1. Example Consider the example programs denoted in Figure 10.2. While we acknowledge that these programs could also be realized using structured control-flow statements only, they nonetheless showcase the challenges of supporting unstructured control-flow statements. The program in Figure 10.2a contains a break-statement in a loop. We can see that the if-statement at line 9 is reached in any iteration it_0, such that both (i) the loop condition check at line 7 holds for all iterations it with $it \leq it_0$, and (ii) the branching condition at line 9 does not hold in any iteration it with $it < it_0$. The program in Figure 10.2b contains a loop with a continue-statement. The assignment of the program variable `sum` at line 13 is reached in any iteration it_0 of the loop, such that both (i) the loop condition check at line 7 holds for all iterations it with $it \leq it_0$, and (ii) the branching condition at line 9 does not hold in iteration it_0. The program in Figure 10.2c contains two goto-statements and a corresponding label `end`. We can see that the expensive computation at line 18 is only reached, if both the branching conditions at line 7 and line 12 do not hold. The program in Figure 10.2d contains an early-return-statement at line 12, and a standard return-statement at line 16. We can see that the return-statement at line 16 is only reached, if the branching condition at line 10 does not hold in any iteration before the iteration where the loop condition check at line 8 does not hold for the first time. □

Let us now consider *forward gotos*, restricted instances of gotos, which do neither (i) introduce additional recursion, nor (ii) jump into contexts.

10.2. Definition For two statements $\mathtt{s}_1$ and $\mathtt{s}_2$ define that s_2 *occurs later than* s_1 *in the program tree*, if there exists a context with two statements $\mathtt{s}_1'$,$\mathtt{s}_2'$ in the program tree,

[3] More precisely, we would need to add skip-statements at the end of both while-statements and functions, so that there is a *statement* (and not only a location) where a continue-statement resp. an early-return-statement can jump to.

```
1  func main()
2  {
3    const Int[] a;
4    const Int len;
5    Int x;
6
7    for (Int i=0; i<len; i=i+1)
8    {
9      if (a[i] == x)
10     {
11       break;
12     }
13   }
14   return i;
15 }
```

(a) Program with a break-statement.

```
1  func main()
2  {
3    const Int[] a;
4    const Int len;
5    Int sum = 0;
6
7    for (Int i=0; i<len; i=i+1)
8    {
9      if (a[i] < 0)
10     {
11       continue;
12     }
13     sum = sum + a[i];
14   }
15   return sum;
16 }
```

(b) Program with a continue-statement.

```
1  func main()
2  {
3    const Int x;
4    const Int y;
5    Int z;
6
7    if (x<0)
8    {
9      z = 0;
10     goto end;
11   }
12   if (x == 1)
13   {
14     z = 1;
15     goto end;
16   }
17   // expensive comp. of z
18   ...
19 end:
20   return 2*x + y + 3*z;
21 }
```

(c) Program with goto-statements.

```
1  func main()
2  {
3    const Int[] a;
4    const Int len;
5    Int x;
6
7    Int i = 0;
8    while (i < len)
9    {
10     if (a[i] == x)
11     {
12       return i;
13     }
14     i = i + 1;
15   }
16   return len;
17 }
```

(d) Program with early-return-statements.

Figure 10.2: Programs containing unstructured control flow.

such that (i) s_2' follows (not necessarily directly) s_1', (ii) s_1' is s_1 itself or a transitive parent of s_1, and (iii) s_2' is s_2 itself or a transitive parent of s_2. A goto-statement `s` in a program is then called a *forward goto*, if it jumps to a statement `l`, such that `l` occurs later in the program tree than `s`.

Forward gotos include break-statements, continue-statements, and early-return-statements as special-cases,[4] and arguably cover almost all real-world usages of gotos.[5] It would therefore make sense to *generalize trace logic to support forward gotos*. We conjecture that this is possible. One research question is how to define the symbol n_w in the presence of forward gotos. If there are no unstructured control-flow statements, then n_w denotes both (i) the first iteration, where the loop condition does not hold anymore, and (ii) the iteration where the loop is exited. But in the presence of forward gotos, these two notions do not necessarily coincide anymore. One could keep notion (i) and introduce for each break statement in the loop an additional symbol, to denote the first iteration where that break statement is reached. Or, one could keep notion (ii) by using a single symbol n_w for any loop w to denote the first iteration where either the loop-condition does not hold or one of the break statements is reached. A related research question is how the presence of forward gotos influences inductive reasoning and the applicability of trace lemmas, and whether new trace lemmas are needed to handle forward gotos.

10.2.3 Reasoning - Efficient Support for Difference Logic over Natural Numbers

Trace logic utilizes the background theory of difference logic over natural numbers to model loop iterations. As described at the end of Section 5.2, we support reasoning with this theory inside Vampire by extending the term algebra of natural numbers with an ordering relation and corresponding axioms. While this solution enables Vampire to prove many interesting program properties, it is far from optimal. In particular, the axioms of the ordering relation (denoted in Figure 5.2) cause a substantial blowup of the search space.

It would be interesting to investigate *a principled solution for efficient superposition-based reasoning with difference logic over natural numbers*. Such a solution would not only be useful in the setting of trace logic, but could also help in many other application settings, where we are interested in a notion of discrete time and the order between timepoints. As a starting point, one could try to adapt the ideas from [BG98] to the setting of difference logic over natural numbers.

10.2.4 Reasoning - Industrial Theorem Proving

While developing our trace-logic-based verification framework, we realized that there are two major challenges for efficiently applying Vampire to a given domain.

[4] To make this precise, we would again need to introduce skip-statements at the end of while-statements and functions to capture continue-statements resp. early-return-statements.

[5] The usage of `goto`-statements, which are not forward gotos, could even be considered bad practice.

First, as already discussed earlier, analyzing saturation attempts is difficult. The SatVis tool, introduced in Chapter 8, is a first step towards understanding saturation attempts efficiently, and was invaluable for the development of the techniques presented in this thesis. But if we really want to tame Vampire and fully understand its behavior, we need to develop additional techniques for the efficient analysis of saturation attempts. In particular, we think it would be invaluable to develop (i) a profiling technique to automatically detect any cyclic behavior in a proof attempt, caused by repeated applications of theory axioms and problematic input axioms, (ii) a technique to enable *efficient* comparisons of the sets of clauses derived in two proof attempts, and (iii) a (semi-automatic) proof transformation to translate superposition-proofs into proofs, which can be inspected more efficiently by humans. It should be noted that the development of such techniques would not only require engineering effort, but also new theoretical insights in proof theory and the analysis of saturation attempts.

Secondly, many of the recently developed techniques that contribute to Vampire's efficient reasoning make Vampire's performance *highly unstable* against small changes to the search space. This includes the LRS saturation strategy [RV03], most of the literal selection functions introduced in [HRSV16], the Avatar architecture [Vor14], and several other techniques. We acknowledge that it is reasonable to trade stability against efficiency, if the only goal is to maximize the number of problems from some benchmark set, for which (a portfolio mode of) Vampire is able to find a proof. But if a predictable and reproducible performance is also important, then the instability of the techniques mentioned above is a big concern, as it is nearly impossible to understand whether a certain change to proof search improves the overall performance of Vampire, which means that it is nearly impossible to properly tune Vampire to a given domain. To restore stability, the simplest solution would be to turn off all instability-introducing techniques. Unfortunately, doing so slows down Vampire considerably, and is – at least on the trace logic domain – not an option. To overcome this problem, it might be interesting to develop restricted variants of the instability-introducing techniques mentioned above, which trade the last bit of efficiency for better stability of Vampire's performance.

We think that solutions for the two challenges presented in this subsection would simplify the overall interaction with Vampire tremendously, and in particular could position Vampire as a more interesting option for industrial applications of first-order theorem proving.

Bibliography

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. Deductive software verification – The KeY book. *Lecture Notes in Computer Science*, 10001, 2016.

[AGS14] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Booster: An acceleration-based verification framework for array programs. In *Proceedings of the 12th International Symposium on Automated Technology for Verification and Analysis (ATWA 2014)*, volume 8837 of *LNCS*, pages 18–23. Springer, 2014.

[ALGC12] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. Ufo: A framework for abstraction-and interpolation-based software verification. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012)*, volume 7358 of *LNCS*, pages 672–678. Springer, 2012.

[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.

[AW13] Noran Azmy and Christoph Weidenbach. Computing tiny clause normal forms. In *Proceedings of the 24th International Conference on Automated Deduction (CADE 2013)*, volume 7898 of *LNCS*, pages 109–125. Springer, 2013.

[BB13] Bernhard Beckert and Daniel Bruns. Dynamic logic with trace semantics. In *Proceedings of the 24th International Conference on Automated Deduction (CADE 2013)*, pages 315–329. Springer, 2013.

[BBC+19] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotić, Carsten Varming, and

Blake Whaley. Reachability analysis for AWS-based networks. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV 2019)*, volume 11562 of *LNCS*, pages 231–241. Springer, 2019.

[BBDL+17] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, K. Rustan M. Leino, Jay Lorch, et al. Everest: Towards a verified, drop-in replacement of https. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[BBE+09] John Byrnes, Michael Buchanan, Michael Ernst, Philip Miller, Chris Roberts, and Robert Keller. Visualizing proof search for theorem prover development. In *Proceedings of the 8th International Workshop on User Interfaces for Theorem Provers (UITP 2008)*, volume 226 of *ENTCS*, pages 23 – 38, 2009.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *Proceedings of the 17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011.

[BCK13] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In *Proceedings of the International Symposium on Logical Foundations of Computer Science (LFCS 2013)*, volume 7734 of *LNCS*, pages 29–43. Springer, 2013.

[BDG14] Musard Balliu, Mads Dam, and Roberto Guanciale. Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS 2014)*, pages 1080–1091. ACM, 2014.

[BDW18] Dirk Beyer, Matthias Dangl, and Philipp Wendler. A unifying view on SMT-based software verification. *Journal of Automated Reasoning*, 60(3):299–335, 2018.

[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.

[Ben04] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, pages 14–25. ACM, 2004.

[Bey20] Dirk Beyer. Advances in automatic software verification: SV-COMP 2020. In *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020)*, volume 12079 of *LNCS*, pages 347–367. Springer, 2020.

[BFG+14] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*, pages 193–205. ACM, 2014.

[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.

[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.

[BG98] Leo Bachmair and Harald Ganzinger. Ordered chaining calculi for first-order theories of transitive relations. *Journal of the ACM*, 45(6):1007–1049, November 1998.

[BGML01] Leo Bachmair, Harald Ganzinger, David A. McAllester, and Christopher Lynch. Resolution theorem proving. In *Handbook of Automated Reasoning*, pages 19–99. Elsevier, 2001.

[BGMR15] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, volume 9300 of *LNCS*, pages 24–51. Springer, 2015.

[BLDM11] Maria Paola Bonacina, Christopher A Lynch, and Leonardo De Moura. On deciding satisfiability by theorem proving with speculative inferences. *Journal of Automated Reasoning*, 47(2):161–189, 2011.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

[BN99] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1999.

[BS01] Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In *Proceedings of the 1st International Joint Conference on Automated Reasoning, (IJCAR 2001)*, volume 2083 of *LNCS*, pages 626–641. Springer, 2001.

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[BU18] Bernhard Beckert and Mattias Ulbrich. Trends in relational program verification. In *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, pages 41–58. Springer, 2018.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CC14] Patrick Cousot and Radhia Cousot. Abstract interpretation: Past, present and future. In *Proceedings of the Joint Meeting of the 23rd EACSL Conference on Computer Science Logic (CSL 2014) and the 29th ACM/IEEE Symposium on Logic in Computer Science (LICS 2014)*, pages 1–10. ACM, 2014.

[CCL11] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, pages 105–118. ACM, 2011.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[CEK+15] Véronique Cortier, Fabienne Eigner, Steve Kremer, Matteo Maffei, and Cyrille Wiedling. Type-based verification of electronic voting protocols. In *Proceedings of the 4th International Conference on Principles of Security and Trust (POST 2015)*, volume 9036 of *LNCS*, pages 303–323. Springer, 2015.

[CFK+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Proceedings of the 3rd International Conference on Principles of Security and Trust (POST 2014)*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014.

[CGLM17] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. A type system for privacy properties. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, pages 409–423. ACM, 2017.

[CGLM18] Veronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. Equivalence properties by typing in cryptographic branching protocols. In *Proceedings of the 7th International Conference on Principles of Security and Trust (POST 2018)*, volume 10804 of *LNCS*, pages 160–187. Springer, 2018.

[CGU20] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Verifying array manipulating programs with full-program induction. In *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020)*, volume 12079 of *LNCS*, pages 22–39. Springer, 2020.

[CJGK+18] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT Press, 2018.

[CLQR07] Shaunak Chatterjee, Shuvendu K Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. A reachability predicate for analyzing low-level software. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *LNCS*, pages 19–33. Springer, 2007.

[Coo18] Byron Cook. Formal reasoning about the security of amazon web services. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV 2018)*, volume 10981 of *LNCS*, pages 38–47. Springer, 2018.

[Cru17] Simon Cruanes. Superposition with structural induction. In *Proceedings of the 11th International Symposium on Frontiers of Combining Systems (FroCoS 2017)*, volume 10483 of *LNCS*, pages 172–188. Springer, 2017.

[CSS03] Michael A Colón, Sriram Sankaranarayanan, and Henny B Sipma. Linear invariant generation using non-linear constraint solving. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 420–432. Springer, 2003.

[DDA10] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *Proceedings of the 19th European Symposium on Programming (ESOP 2019)*, volume 6012 of *LNCS*, pages 246–266. Springer, 2010.

[DHK16] Przemysław Daca, Thomas A. Henzinger, and Andrey Kupriyanov. Array folds logic. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016)*, volume 9780 of *LNCS*, pages 230–248. Springer, 2016.

[DHKR11] Alastair F Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *Proceedings of the 18th International Symposium on Static Analysis (SAS 2011)*, volume 6887 of *LNCS*, pages 351–368. Springer, 2011.

[DK20] André Duarte and Konstantin Korovin. Implementing superposition in iProver (system description). In *Proceedings of the 10th International Joint Conference on Automated Reasoning (IJCAR 2020)*, volume 12167 of *LNCS*, pages 388–397. Springer, 2020.

[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.

[DMB07] Leonardo De Moura and Nikolaj Bjørner. Efficient e-matching for SMT solvers. In *Proceedings of the 21st International Conference on Automated Deduction (CADE 2007)*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.

[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 2008)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the 3rd Conference on Theory of Cryptography (TCC 2006)*, volume 3876 of *LNCS*, pages 265–284. Springer, 2006.

[EM13] Fabienne Eigner and Matteo Maffei. Differential privacy by typing in security protocols. In *Proceedings of the 26th Symposium on Computer Security Foundations (CSF 2013)*, pages 272–286. IEEE, 2013.

[FGK+14] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE 2014)*, pages 349–360. ACM, 2014.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI 2002)*, pages 234–245. ACM, 2002.

[FPMG19] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In *Proceedings*

of the 31st International Conference on Computer Aided Verification (CAV 2019), volume 11561 of *LNCS*, pages 259–277. Springer, 2019.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 191–202. ACM, 2002.

[GGK20] Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace logic for inductive loop reasoning. In *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design (FMCAD 2020)*, pages 255–263. TU Wien Academic Press, 2020.

[GHM13] Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for information flow control in Java programs - a practical guide. *Software Engineering 2013 - Workshopband*, 2013.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[GKKV14] Ashutosh Gupta, Laura Kovács, Bernhard Kragl, and Andrei Voronkov. Extensional crisis and proving identity. In *Proceedings of the 12th International Symposium on Automated Technology for Verification and Analysis (ATVA 2014)*, volume 8837 of *LNCS*, pages 185–200. Springer, 2014.

[GKR18] Bernhard Gleiss, Laura Kovács, and Simon Robillard. Loop analysis by quantification over iterations. In *Proceedings of the 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2018)*, volume 57 of *EPiC Series in Computing*, pages 381–399. EasyChair, 2018.

[GKR20] Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption demodulation in first-order theorem proving. In *Proceedings of the 10th International Joint Conference on Automated Reasoning (IJCAR 2020)*, volume 12166 of *LNCS*, pages 297–315. Springer, 2020.

[GKS17] Bernhard Gleiss, Laura Kovács, and Martin Suda. Splitting proofs for interpolation. In *Proceedings of the 26th International Conference on Automated Deduction (CADE 2017)*, volume 10395 of *LNCS*, pages 291–309. Springer, 2017.

[GKS19] Bernhard Gleiss, Laura Kovács, and Lena Schnedlitz. Interactive visualization of saturation attempts in Vampire. In *Proceedings of the 15th International Conference on Integrated Formal Methods (IFM 2019)*, volume 11918 of *LNCS*, pages 504–513. Springer, 2019.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

[GMF+18] Niklas Grimm, Kenji Maillard, Cédric Fournet, Cătălin Hriţcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*, pages 130–145. ACM, 2018.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, 2000.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme i. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[GR09] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An efficient invariant generator. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009)*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.

[GS] Bernhard Gleiss and Martin Suda. Layered clause selection for saturation-based theorem proving. In *Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR 2020)*. Accepted for Publication.

[GS13] Benny Godlin and Ofer Strichman. Regression verification: Proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.

[GS20] Bernhard Gleiss and Martin Suda. Layered clause selection for theory reasoning (short paper). In *Proceedings of the 10th International Joint Conference on Automated Reasoning (IJCAR 2020)*, volume 12166 of *LNCS*, pages 402–409. Springer, 2020.

[GSV18] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Quantifiers on demand. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA 2018)*, volume 11138 of *LNCS*, pages 248–266. Springer, 2018.

[HB12] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT 2012)*, volume 7317 of *LNCS*, pages 157–171. Springer, 2012.

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

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

[HPWW13] Thomas Hillenbrand, Ruzica Piskac, Uwe Waldmann, and Christoph Weidenbach. From search to computation: Redundancy criteria and simplification at work. In *Programming Logics: Essays in Memory of Harald Ganzinger*, volume 7797 of *LNCS*, pages 169–193. Springer, 2013.

[HR18] Hossein Hojjat and Philipp Rümmer. The ELDARICA horn solver. In *Proceedings of the 18th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2018)*, pages 1–7. IEEE, 2018.

[HRSV16] Kryštof Hoder, Giles Reger, Martin Suda, and Andrei Voronkov. Selecting the selection. In *Proceedings of the 8th International Joint Conference on Automated Reasoning (IJCAR 2016)*, volume 9706 of *LNCS*, pages 313–329. Springer, 2016.

[HV11] Krystof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE 2011)*, volume 6803 of *LNCS*, pages 299–314. Springer, 2011.

[Kam80] Sam Kamin. Two generalizations of the recursive path ordering. *Unpublished manuscript*, 1980.

[KB83] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

[KFG20] Naoki Kobayashi, Grigory Fedyukovich, and Aarti Gupta. Fold/unfold transformations for fixpoint logic. In *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020)*, volume 12079 of *LNCS*, pages 195–214. Springer, 2020.

[KGC16] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[KHE17] Hyoukjun Kwon, William Harris, and Hadi Esmaeilzadeh. Proving flow security of sequential logic via automatically-synthesized relational invariants. In *Proceedings of the 30th IEEE Symposium on Computer Security Foundations (CSF 2017)*, pages 420–435. IEEE, 2017.

[KKRV16] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. The Vampire and the FOOL. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2016)*, pages 37–48. ACM, 2016.

[KKV18] Evgenii Kotelnikov, Laura Kovács, and Andrei Voronkov. A FOOLish encoding of the next state relations of imperative programs. In *Proceedings of the 9th International Joint Conference on Automated Reasoning (IJCAR 2018)*, volume 10900 of *LNCS*, pages 405–421. Springer, 2018.

[KMV11] Laura Kovács, Georg Moser, and Andrei Voronkov. On transfinite knuth-bendix orders. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE 2011)*, volume 6803 of *LNCS*, pages 384–399. Springer, 2011.

[KRV17] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to terms with quantified reasoning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, pages 260–270. ACM, 2017.

[KV09a] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE 2009)*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009.

[KV09b] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In *Proceedings of the 22nd International Conference on Automated Deduction (CADE 2009)*, volume 5663 of *LNCS*, pages 199–213. Springer, 2009.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

[LHKR12] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. SYMDIFF: A language-agnostic semantic Diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012)*, volume 7358 of *LNCS*, pages 712–717. Springer, 2012.

[LM09] K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC 2009)*, pages 615–622. ACM, 2009.

[Löc06a] Bernd Löchner. Things to know when implementing KBO. *Journal of Automated Reasoning*, 36(4):289–310, 2006.

[Löc06b] Bernd Löchner. Things to know when implementing LPO. *International Journal on Artificial Intelligence Tools*, 15(01):53–79, 2006.

[LRR14] Tomer Libal, Martin Riener, and Mikheil Rukhaia. Advanced proof viewing in ProofTool. In *11th Workshop on User Interfaces for Theorem Provers (UITP 2014)*, pages 35–47, 2014.

[Mos09] Michał Moskal. Programming with triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT 2009)*, pages 20–29. ACM, 2009.

[NN93] Pilar Nivela and Robert Nieuwenhuis. Saturation of first-order (constrained) clauses with the Saturate system. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA 2001)*, volume 2051 of *LNCS*, pages 436–440. Springer, 1993.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning*, pages 371–443. Elsevier, 2001.

[NS16] Kedar S. Namjoshi and Nimit Singhania. Loopy: Programmable and formally verified loop transformations. In *Proceedings of the 23rd International Symposium on Static Analysis (SAS 2016)*, volume 9837 of *LNCS*, pages 383–402. Springer, 2016.

[OMW76] R Overbeek, J McCharen, and Larry Wos. Complexity and related enhancements for automated theorem-proving programs. *Computers & Mathematics with Applications*, 2(1):1–16, 1976.

[PG86] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17 – 139, 2004. Original version: University of Aarhus Technical Report DAIMI FN-19, 1981.

[PY14] Nimrod Partush and Eran Yahav. Abstract semantic differencing via speculative correlation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014)*, pages 811–828. ACM, 2014.

[RBF18] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2018)*, pages 112–131. Springer, 2018.

[RBSV16] Giles Reger, Nikolaj Bjorner, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In *Proceedings of the 2nd Global Conference on Artificial Intelligence (GCAI 2016)*, volume 41 of *EPiC Series in Computing*, pages 39–52. EasyChair, 2016.

[RES10] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology (AMAST 2010)*, volume 6486 of *LNCS*, pages 142–162. Springer, 2010.

[Rey16] Andrew Reynolds. Conflicts, models and heuristics for quantifier instantiation in SMT. In *Proceedings of the 3rd Vampire Workshop (Vampire 2016)*, EPiC Series in Computing, pages 1–15. EasyChair, 2016.

[RK15] Andrew Reynolds and Viktor Kuncak. Induction for SMT solvers. In *Proceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2015)*, volume 8931 of *LNCS*, pages 80–98. Springer, 2015.

[RL17] Pritom Rajkhowa and Fangzhen Lin. VIAP - Automated system for verifying integer assignment programs with loops. In *Proceedings of the 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2017)*, pages 137–144. IEEE, 2017.

[Rot16] Frederik Rothenberger. Integration and analysis of alternative SMT solvers for software verification. Master's thesis, ETH Zurich, 2016.

[RR18] Giles Reger and Martin Riener. What is the point of an SMT-LIB problem? In *Proceedings of the 16th International Workshop on Satisfiability Modulo Theories (SMT 2018)*, 2018.

[RS12] Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *Proceedings of the 2012 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2012)*, pages 555–574. ACM, 2012.

[RS17] Giles Reger and Martin Suda. Set of support for theory reasoning. In *Proceedings of the IWIL 2017 Workshop and LPAR-21 Short Presentations*, volume 1 of *Kalpa Publications in Computing*. EasyChair, 2017.

[RSV15] Giles Reger, Martin Suda, and Andrei Voronkov. Playing with AVATAR. In *Proceedings of the 25th International Conference on Automated Deduction (CADE 2015)*, volume 9195 of *LNCS*, pages 399–415. Springer, 2015.

[RSV16] Giles Reger, Martin Suda, and Andrei Voronkov. New techniques in clausal form generation. *Proceedings of the 2nd Global Conference on Artificial Intelligence (GCAI 2016)*, 41:11–23, 2016.

[RSV18] Giles Reger, Martin Suda, and Andrei Voronkov. Unification with abstraction and theory instantiation in saturation-based reasoning. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2018)*, volume 10805 of *LNCS*, pages 3–22. Springer, 2018.

[RTDM14] Andrew Reynolds, Cesare Tinelli, and Leonardo De Moura. Finding conflicting instances of quantified formulas in SMT. In *Proceedings of the 14th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2014)*, pages 195–202. IEEE, 2014.

[RV01] Alexandre Riazanov and Andrei Voronkov. Splitting without backtracking. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 611–617. Morgan Kaufmann, 2001.

[RV03] Alexandre Riazanov and Andrei Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1-2):101–115, 2003.

[RV19] Giles Reger and Andrei Voronkov. Induction in saturation-based proof search. In *Proceedings of the 27th International Conference on Automated Deduction (CADE 2019)*, volume 11716 of *LNCS*, pages 477–494. Springer, 2019.

[RWB+17] Andrew Reynolds, Maverick Woo, Clark W. Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. Scaling up DPLL(T) string solvers using context-dependent simplification. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV 2017)*, volume 10427 of *LNCS*, pages 453–474. Springer, 2017.

[Sch02] Stephan Schulz. E - a brainiac theorem prover. *AI Communications*, 15(2-3):111–126, 2002.

[Sch13] Stephan Schulz. System description: E 1.8. In *Proceedings of the 19th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2013)*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[SCV19] Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, higher, stronger: E 2.3. In *Proceedings of the 27th International Conference on Automated Deduction (CADE 2019)*, volume 11716 of *LNCS*, pages 495–507. Springer, 2019.

[SD08] Eric Whitman Smith and David L. Dill. Automatic formal verification of block cipher implementations. In *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2008)*, pages 1–7. IEEE, 2008.

[SD16] Marcelo Sousa and Isil Dillig. Cartesian Hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 57–69. ACM, 2016.

[SDL18] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. Verified three-way program merge. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):165:1–165:29, 2018.

[Seb07] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(3-4):141–224, 2007.

[SG18] Martin Suda and Bernhard Gleiss. Local soundness for QBF calculi. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT 2018)*, volume 10929 of *LNCS*, pages 217–234. Springer, 2018.

[SHK+16] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43th ACM-SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pages 256–270. ACM, 2016.

[SM03] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[SM16] Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In *Proceedings of the 8th International Joint Conference on Automated Reasoning (IJCAR 2016)*, volume 9706 of *LNCS*, pages 330–345. Springer, 2016.

[SP+78] Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis*. New York University. Institute of Mathematical Sciences, 1978.

[SRV01] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Term indexing. In *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier, 2001.

[SSCA13] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN*

International Conference on Object Oriented Programming Systems Languages & Applications, (OOPSLA 2013), pages 391–406. ACM, 2013.

[SST14] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In *Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR 2014)*, volume 6806 of *LNCS*, pages 367–373. Springer, 2014.

[STL11] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for LLVM. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, volume 6806 of *LNCS*, pages 737–742. Springer, 2011.

[Sud20] Martin Suda. Aiming for the goal with SInE. In *Proceedings of the 5th and 6th Vampire Workshops (Vampire 2018 and Vampire 2019)*, volume 71 of *EPiC Series in Computing*, pages 38–44. EasyChair, 2020.

[Sut07] Geoff Sutcliffe. TPTP, TSTP, CASC, etc. In *Proceedings of the 2nd International Symposium on Computer Science in Russia (CSR 2007)*, volume 4649 of *LNCS*, pages 367–373. Springer, 2007.

[Sut16] Geoff Sutcliffe. The CADE ATP system competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[Sut17] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. from CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, Feb 2017.

[Tan18] Ole Tange. *GNU Parallel 2018*. Ole Tange, March 2018.

[Vor01] Andrei Voronkov. Algorithms, datastructures, and other issues in efficient automated deduction. In *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR 2001)*, volume 2083 of *LNCS*, pages 13–28. Springer, 2001.

[Vor14] Andrei Voronkov. AVATAR: The architecture for first-order theorem provers. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2014)*, volume 8559 of *LNCS*, pages 696–710. Springer, 2014.

[WCD+19] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015-2018. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):221–259, 2019.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In *Proceedings of the 22nd International Conference on Automated Deduction (CADE 2009)*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.

[Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. In *Handbook of Automated Reasoning*, pages 1965–2013. Elsevier, 2001.

[WG12] William M Waite and Gerhard Goos. *Compiler construction*. Springer, 2012.

[WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT 2014)*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.

[WRC65] Lawrence Wos, George A Robinson, and Daniel F Carson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM*, 12(4):536–541, 1965.

[WTRB20] Uwe Waldmann, Sophie Tourret, Simon Robillard, and Jasmin Blanchette. A comprehensive framework for saturation theorem proving. In *Proceedings of the 10th International Joint Conference on Automated Reasoning (IJCAR 2020)*, volume 12166 of *LNCS*, pages 316–334. Springer, 2020.

[WW08] Christoph Weidenbach and Patrick Wischnewski. Contextual rewriting in SPASS. In *Proceedings of the 1st International Workshop on Practical Aspects of Automated Reasoning (PAAR 2008)*, 2008.

[ZHH17] Qi Zhou, David Heath, and William Harris. Completely automated equivalence proofs. *CoRR*, abs/1705.03110, 2017.