

Towards Scalable Fully Automatic Program Verification

by

Hari Govind V K

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2024

© Hari Govind V K 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Kenneth L. McMillan Professor, Dept. of Computer Science, University of Texas at Austin
Supervisor	Arie Gurfinkel Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Member	Werner Dietl Associate Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Member	Vijay Ganesh Professor, School of Computer Science, Georgia Institute of Technology
Internal-external Member	Richard Trefler Associate Professor, David R. Cheriton School of Computer Science, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Contents and results presented in this thesis are collaborations with my supervisor, Professor Arie Gurfinkel, together with our collaborators. Chp. 2 was done in collaboration with Dr. YuTing Chen from Chalmers University and Professor Sharon Shoham from Tel Aviv University. Chp. 3 was done in collaboration with Professor Sharon Shoham. Finally, Chp. 4 and Chp. 5 were done in collaboration with Dr. Isabel García Contreras and Professor Sharon Shoham.

Abstract

Formal verification is a rigorous way to establish software quality. After more than 50 years of research, we are at a point where formal verification is starting to be used in industrial applications. To encourage more adoption, we need to automate formal verification techniques as much as possible so that even users who are not familiar with formal verification can use it to ensure software quality. Many automatic verification tasks are reducible to the problem of satisfiability checking of Constrained Horn Clauses (CHCs) modulo background theories. The list includes safety verification of imperative and functional programs, regression verification, static analysis, refinement type inference, and many more. Motivated by the number of applications, we explore advancements to CHC solving.

We make three contributions to improving the state of the art in CHC solving. First, we study the problem of lemma generalization inside CHC solvers. State of the art CHC solvers construct solutions to CHCs by learning several lemmas, each of which block parts of the search space, and together form a complete solution. However, the generalization strategies that CHC solvers employ during lemma learning are *local*: they are unaware of the solution that the CHC solver is constructing. This causes CHC solvers to diverge and worse, be susceptible to syntactic changes in the input. We mitigate the effects of local generalization by identifying patterns in lemmas learned during CHC solving and enforcing course correction during CHC solving via a process called global guidance. We instantiate global guidance inside the SPACER CHC solver and show that it improves SPACER on a variety of benchmarks and makes SPACER agnostic to local generalization strategies.

In the second part of the thesis, we look at CHC solving in the context of safety verification of programs manipulating user defined recursive datatypes like lists and trees. The invariants of such programs typically require recursive functions as well. This domain is particularly hard for CHC solvers because the presence of recursive functions make even validating solutions to CHCs undecidable. We propose a new approach that sidesteps the undecidability by abstracting recursive functions using uninterpreted functions when validating solutions. We show the typical approach to overcome this undecidability, that of eliminating recursive functions by encoding them as a set of CHCs, loses solutions and is therefore infeasible for many CHC solvers.

In the third part of the thesis, we go deeper inside CHC solvers and propose algorithmic improvements to the solutions to two important subproblems encountered during CHC solving: modular solving and approximations of quantifier elimination.

Acknowledgments

First and foremost, I extend my deepest gratitude to Professor Arie Gurfinkel, my PhD advisor. He has guided me in framing research problems, coding, understanding academic papers, delivering presentations, and practically everything I know about research. His dedication to mentoring me, often at unconventional hours, and his patient correction of my mistakes have been invaluable. His patience, expertise, and passion for research are truly inspirational.

I am sincerely grateful to Professor Sharon Shoham for collaborating with me on almost all my papers. She has always been patient with my drafts, providing clear and insightful feedback, helping me identify the core issues (the UNSAT core, if you will). A single meeting with Sharon has always been more enlightening than several days of solo work.

I thank Professor Ken McMillan for agreeing to be my external examiner, reviewing my thesis, and offering insightful comments. I also extend my thanks to my PhD examining committee: Professors Vijay Ganesh, Richard Trefler, and Werner Dietl. Their reviews and discussions have been a source of encouragement throughout my PhD journey.

Special thanks to Dr. Christoph Sticksele for mentoring me during my internships at MathWorks. My time at MathWorks provided me with invaluable industrial research experience, and Christoph ensured that my internships were both informative and enjoyable. I especially appreciate the extraordinary efforts Christoph made to facilitate my first internship against all odds during the COVID-19 pandemic.

I am grateful to the many friends who helped me flourish during my PhD. To my first Waterloo friends, Jakub Kuderski and Nham Le, with whom I enjoyed everything from board games to barbecues. To my lab mates, Siddharth Priya, Chunxiao (Ian) Li, Isabel García Contreras, Joseph Tafese, Yusen Su, and many others, who made the lab a lively and enjoyable place with their stimulating discussions. To my football mates, Mehdi Moslemi Aqdam and Mary Saleh, who shared my passion for the sport. And to my climbing mates, Sharat Ibrahimpur, Julie Messier, Hemant Saxena, Priya Soundararajan, and Nathaniel Harms, who shared my love for problem-solving.

A special thanks to my sister, Lekshmi G K, for always being my guiding light. As in so many other things, I would not have embarked on the journey to Canada without your reassurance. I also thank my brother, Deepak C U, for his unwavering support throughout my PhD.

I express my heartfelt gratitude to my wife, Parvathy V J, for her love, trust, and unconditional support.

Finally, I thank my Mother and Father for their endless love, which I feel every day, even from opposite ends of the world.

Dedication

This thesis is dedicated to my parents who continue to sacrifice so much for me.

Table of Contents

List of Figures	x
List of Tables	xii
List of Algorithms	xiii
Introduction	1
1 Preliminaries	6
1.1 Logic	6
1.2 Theories	7
1.2.1 Equality and Uninterpreted Functions	7
1.2.2 Linear Integer Arithmetic	8
1.2.3 Algebraic Data Types	9
1.2.4 Recursively Defined Functions	10
1.3 Quantifier elimination	11
1.4 Interpolation	12
1.5 Transition Systems	12
1.6 IC3/PDR	13
1.7 Constrained Horn Clauses	14
1.8 SPACER	15
1.8.1 SPACER for linear CHCs	15
1.8.2 SPACER for linear and non-linear CHCs	18
2 Global Guidance For Linear Arithmetic	21
2.1 Pitfalls of Local reasoning	23
2.2 Global guidance rules	25
2.3 Global guidance for Linear Integer Arithmetic	27
2.3.1 Lemma selection	27
2.3.2 Subsume rule for LIA	29
2.3.3 Concretize rule for LIA	34

2.3.4	Conjecture rule for LIA	36
2.3.5	Putting it all together	37
2.3.6	Global Guidance for Linear Rational Arithmetic	39
2.4	Evaluation	39
2.4.1	Comparison with SPACER.	41
2.4.2	Comparison with LINEARARBITRARY.	43
2.4.3	Effectiveness of individual Global Guidance rules	44
2.5	Related Work	44
2.6	Conclusion	47
3	CHC Solvers For Programs Manipulating Data Types	48
3.1	Overview	51
3.2	CHCs modulo ADTs and Catamorphisms	57
3.3	Reducing CHCs modulo Catamorphisms to CHCs	58
3.4	Introducing RDF Abstractions into CHCs	63
3.5	CHCs modulo RDF vs CHCs over RDF abstractions	66
3.6	Solving CHC Modulo ADT and RDF	68
3.7	Implementation and Evaluation	72
3.8	Conclusion	75
4	Fast Approximations Of Quantifier Elimination	76
4.1	An Egraph of a Formula	79
4.2	Extracting Formulas from Egraphs	80
4.3	Quantifier Reduction	85
4.4	Model Based Projection Using QEL	96
4.5	Evaluation	101
4.6	Conclusion	103
5	Speculative SAT Modulo SAT	104
5.1	Motivating examples	106
5.2	Speculative SAT Modulo SAT	107
5.2.1	Sat Modulo Sat	107
5.2.2	Speculative Sat Modulo Sat	109
5.3	Validation and interpolation	113
5.3.1	DRUP proofs for modular SAT	113
5.3.2	Interpolation	117
5.4	Heuristics for guiding SPECSMS	119
5.5	Implementation and Validation	120
5.6	Conclusion and Future Work	121

6 Conclusions	123
Bibliography	125

List of Figures

1.1	Example egraph of $\varphi_{1.1}$.	8
1.2	An example of a simple program.	17
2.1	Pitfalls of local reasoning	23
2.2	Best configurations on LIA benchmarks: GSPACER versus SPACER.	41
2.3	Comparing GSPACER with different interpolation tactics on LIA benchmarks.	42
2.4	Comparing GSPACER with just the Subsume rule (x-axis) against GSPACER with all three rules together (y-axis).	45
3.1	A Dafny program that manipulates a list.	49
3.2	Overview of our procedure for CHCs modulo ADTs and RDFs	52
3.3	RDF abstractions for solving CHCs modulo ADTs and RDFs	53
3.4	Comparing $\mathcal{A}$ -solutions of different abstractions of CHCs	67
4.1	Example egraph of $\varphi_{4.1}$.	79
4.2	Different egraph interpretations for $\varphi_{4.2}$.	79
4.3	Egraphs of $\varphi_{4.3}$ with G_{repr} .	82
4.4	Egraphs including G_{repr} of $\varphi_{4.4}$.	87
4.5	Egraph implementation of MBP rules from [70]	97
5.1	Speculative SMS automata	109
5.2	A modular DRUP proof	114

List of Tables

2.1	Number of LIA benchmarks	39
2.2	Number of LRA benchmarks	39
2.3	Comparison between SPACER and GSPACER on LIA benchmarks from CHC-COMP.	40
2.4	Comparison between SPACER and GSPACER on LRA benchmarks from CHC-COMP.	42
2.5	Comparison with LARB using CHC-COMP benchmarks.	44
2.6	Effectiveness of individual rules of GSPACER on CHC-COMP LIA instances.	44
3.1	Performance of RACER on RUST-BENCH	72
3.2	Performance of RACER on LEON-BENCH	72
4.1	Comparison between Z3EG and Z3 on SMT-COMP instances	102
4.2	Comparison between Z3EG and Z3 on handcrafted benchmarks	102
4.3	Performance of Z3EG on CHC benchmarks	103
5.1	Validating SPECSMS	120

List of Algorithms

1	SPACER algorithm as a set of guarded commands	15
2	The SPACER algorithm for linear CHCs.	16
3	The SPACER algorithm for linear and non-linear CHCs.	19
4	Global guidance rules for SPACER.	25
5	Implementation of Subsume rule in LIA	30
6	An implementation of the Concretize rule in LIA.	35
7	GSPACER for LIA.	37
8	The RACER algorithm	69
9	Producing formulas from an egraph.	81
10	QEL – Quantifier reduction using egraphs.	86
11	Find definitions maximizing groundness.	88
12	Refining repr and building core.	94
13	MBP-QEL: an MBP using QEL	100
14	The REASON method in modular SAT solvers inside SPECSMS	112
15	Trimming a modular DRUP proof.	116
16	Interpolating a modular DRUP proof.	116

Introduction

Promise and practice of formal verification Ensuring software quality is an important task. Quality can mean anything from ensuring that software does not crash to ensuring that the software is free of security vulnerabilities. Formal verification is the rigorous process of mathematically proving the correctness of software with respect to a set of specifications. Regardless of what quality means, formal verification is a promising way to ensure software quality. For a long time, formal verification has been considered impractical. However, owing to a lot of research in better algorithms, formal verification is starting to get adopted in the industry. To name just a few, the IVy tool and language [83] is used to implement distributed protocols and automatically verify their properties, the Dafny [76] programming language is used to write provably correct imperative code, and project Everest [91] focuses on building and deploying formally verified communication protocols. Formal methods is used to verify access control permissions on Amazon Web Services, to prevent unintended data access [6], improve general quality of code by writing proofs along side code [21], and even expose side channel attacks by statically analyzing execution times across different control flows [5]. The Infer tool is used to find bugs and security vulnerabilities in each commit to their codebase [30].

While some of these techniques are automated, most are not, meaning that the developer has to write proofs, along side code, in order for them to work. Since proofs are complicated, they are usually written by expert formal verification engineers. This is a major hurdle to widespread adoption of formal verification tools. This thesis focuses on scaling fully automatic tools for formal verification.

To better enable development of automatic program verification techniques, researchers have used intermediate representations to split the verification pipeline into two. The first phase encodes the program verification task as a problem in a suitable intermediate representation and the second phase uses fully automatic tools to solve the problem in the intermediate representation. This way, researchers can focus on constructing automated tools without worrying about the unimportant specifics of the program at hand (for example, syntax of the programming language).

CHCs as an intermediate representation Constrained Horn Clauses (CHCs) are one such intermediate representation. CHCs are a fragment of first order logic that is expressive enough to encode safety verification of programs while not involving any quantifier alternations. A salient feature of CHCs is that they use background SMT theories to express common programming constructs. SMT theories are rich enough to express operators like $+$, $-$, $\ll$, to high level datastructures like lists and trees. CHC solvers use SMT solvers to efficiently reason about these constructs. The satisfiability of CHCs translates to safety of the encoded program. A solution to CHCs gives loop invariants and function summaries that prove safety. On the other hand, unsatisfiability of a CHC is a witness of safety violation. The proof of unsatisfiability of CHC gives a feasible execution trace of the program that violates specification.

CHCs have been used in the verification of recursive sequential programs written in C/C++ [55], Java [68], Rust [79], and many other languages [14]. The Solidity programming language for smart contracts ships with a formal verification engine that translates smart contracts, models of environment, and their specifications into CHCs [4]. CHCs have also been used to prove equivalence of code before and after refactoring [27]. For hyperproperty verification, CHCs can encode both the problem of finding a good alignment for self composition as well as finding the invariant for the composition [62]. CHCs are beginning to gain traction outside automated program verification as well. In program synthesis, CHCs have been proposed as a standard format for expressing both the syntax and semantics of the program to be synthesized [69], and even to synthesize CHC solvers [51]. CHCs have also been used as intermediate representation for many static analyzers [84] where they enable modular, context sensitive, and incremental analysis [46]. Even refinement type inference for functional programs benefit from using CHCs as intermediate representation [93]. In short, usage of CHC solvers for automated program verification is akin to usage of SAT solvers for solving NP complete problems. Advances to CHC solvers benefit all problems that are reducible to CHC solving.

Spacer SPACER [71] is a state of the art CHC solver. It is the default solver for the horn fragment in the Z3 SMT solver [85] and is a regular winner at CHC-Comp [56]. SPACER, based on the popular IC3/PDR [17], [33] model checking algorithm, incrementally constructs a solution to CHCs by generating *proof obligations* and blocking them and generalizing, to learn lemmas. The ideas presented in this thesis are implemented on top of SPACER. But the ideas are more generally applicable to other CHC solvers and model checkers as well.

Contributions This thesis advances the state of the art in CHC solving by making algorithmic improvements to the solving procedure and broadening their applicability.

In Chp. 2, we study the problem of lemma generalization inside CHC solvers. State of the art CHC solvers construct solutions to CHCs by learning several lemmas, each of which block a part of a search space, and putting them together to form a complete solution. It is crucial that each of these lemmas are general enough to block large portions of the search space. CHC solvers typically rely on interpolation algorithms to do this generalization. However, this generalization is *local*: interpolants are decided by heuristics implemented inside an SMT solver, and are unaware of the solution the CHC solver is constructing. This has several negative consequences. Sometimes it makes the solver diverge because of insufficient or over generalization. Other times, it makes the solver susceptible to small syntactic changes in the input. Sometimes a change as simple as replacing $a = 0 \wedge b = 0$ with $a = 0 \wedge b = a$ in some part of a CHC changes the proof constructed by an SMT solver, thereby changing the interpolant and causing the CHC solver to diverge. We give three examples to illustrate different ways in which local generalization causes CHC solvers to diverge. In all three cases, CHC solvers diverge by learning lemmas similar to each other, all of which are good individually but do not add up to a complete solution. To mitigate this, we introduce a framework for heuristically identifying when solvers diverge and course correcting them. Our key insight is to look for patterns in lemmas learned by the solver and introduce pattern based rules to either generalize lemmas based on the global state of the solver or provide an alternate proof direction for the solver. We implement this idea, called *Global Guidance*, on top of SPACER. We demonstrate its effectiveness empirically by running on a wide variety of benchmarks and under different generalization strategies.

Global guidance was very successful. It was accepted into the main branch of Z3 and is being used by many users of SPACER. We had several anecdotal reports of its success from these users. Global guidance also had a huge impact when we employed SPACER for the design verification of industrial scale Simulink models. Inspired by its effectiveness, we instantiated global guidance for other background theories as well. For example, global guidance was effective on practical hardware verification and equivalence checking benchmarks in the theory of Fixed Size Bit Vectors [65]. We also used global guidance to reimplement the lemma generalization strategy for theory of arrays presented in [57].

While global guidance was implemented for these popular background theories, SPACER was being used with other background theories as well. The Solidity model checker was using SPACER for model checking smart contracts where the environment was modelled using a combination of arrays and lists [3]. There was work on safety verification of imperative programs where memory was modelled using a user-defined (recursive) heap datastructure [36]. Similarly, there was interest in modelling datastructures like lists as-is in CHCs [79].

Motivated by these applications, we look into safety verification of programs manipulating recursively defined datatypes (ADTs like lists) in Chp. 3. Such datastructures require recursively defined functions (RDFs) to summarize their properties (like length of a list). Therefore, the CHC encoding of verification tasks contain recursive functions as well. However, satisfiability checking of even quantifier free formulas containing RDFs is undecidable. This poses a problem for CHC solvers because validating solutions to CHCs fall into this fragment. Furthermore, SMT solvers are not well suited for this fragment because reasoning about RDFs require coming up with inductive summaries (e.g. proving that length of a list is positive). Since CHC solvers are designed to generate inductive summaries, the common approach to solve CHC over ADTs and RDFs is to remove RDFs from the background theory by encoding RDFs using CHCs. We formalize this approach and show that it loses solutions. Therefore, this approach is not feasible for solvers that solve CHCs by *constructing* a solution. We propose a CHC solving procedure that keeps around both RDFs and their CHC encoding. Our approach carefully abstract RDFs using uninterpreted functions (UF) whenever it makes queries to the underlying SMT solver, thereby sidestepping the problem of undecidability of RDFs. Just like all abstractions, our abstractions also lead to spurious counterexamples. To maintain soundness, our solution automatically refines the abstraction when spurious counterexamples are encountered. We implement an algorithm that smartly integrates this abstraction refinement loop inside the SPACER algorithm. We evaluate the algorithm on a suite of benchmarks encoding safety verification of imperative and functional programs.

In the third part of the thesis (Chp. 4 and 5), we look deeper into the SPACER algorithm and propose algorithmic improvements to solutions of two important subproblems that SPACER solves.

The first subproblem we look at is Quantifier Elimination (qelim). A recurring step in SPACER is computing predecessor *proof obligations* by projecting facts about the head of a CHC to facts about its tail. To achieve this, SPACER uses a 2 step approximation of qelim. In the first step, SPACER employs a cheap and incomplete procedure for qelim based on the well known variable substitution rule. For the variables left after this step, SPACER computes a model-based under-approximation of qelim called Model Based Projection (MBP). In Chp. 4 we improve both these steps. We use the egraph datastructure [88] to implement a relatively-complete qelim, called QEL. We show that QEL eliminates all variables that are equivalent to ground terms. We then implement SPACER’s MBP algorithms on top of QEL. We show that our QEL based MBP algorithm computes a less under-approximate MBP (i.e. is closer to full qelim). Even though an under-approximation of qelim is just one step inside SPACER, we observed that with QEL and MBP based QEL, SPACER converged on significantly more instances. Our implementation has been integrated into the main branch of Z3.

The algorithm is used not just inside SPACER but also inside the decision procedure for Quantified Satisfiability [15].

The second subproblem we look at is *modular* solving where a bounded model checking query is split up into multiple modules, each of which is solved by a (virtually) separate solver. It is at the heart of IC3/PDR as well as all model checking algorithms based off it, including SPACER. Modular solving determines the shape of inductive invariant that IC3/PDR learns as well as how quickly the algorithm converges. Briefly, modular solvers find a full satisfying assignment to one of the modules, projects the solution to be over shared symbols, and checks if the projection is extendable to be a full satisfying assignment to the other module. If not, modular solvers block the satisfying assignment by learning a lemma in the first module. Modular solvers have the advantage that the conjunctions of all the lemmas learned, form a CNF interpolant. The disadvantage with modular solving however, is that it is not as efficient as monolithic solving. In Chp. 5 we bridge this gap by extending modular solving with *speculation*: the ability to switch between solving modules at any point during solving. We show that speculation improves modular solving and study the shape of interpolants generated by the speculative algorithm. We implement the new modular SAT solver as an extension to the SAT solver inside Z3 and empirically validate our claims.

While this thesis presents a series of individual improvements to CHC solving, they also pave the way for a new kind of CHC solver that works over an abstract core theory and lazily applies theory reasoning. We discuss such future possibilities in Chp. 6.

Chapter 1

Preliminaries

In this chapter we give necessary background and set up notation required for the rest of this thesis.

1.1 Logic

We work in standard many-sorted First Order Logic with equality, i.e., we assume that we have the equality symbol ($\approx$) for all sorts. Terms and formulas are defined over a signature Σ which consists of predicate and function symbols, each with a designated non-negative arity and sorts. We refer to zero-ary function and predicate symbols as constants. A term t is either a (sorted) variable x , or a function symbol applied to terms, $f(t_1, \dots, t_n)$, respecting the arity and sorts of f . An atomic formula is an equality applied to terms of the same sort, $t_1 \approx t_2$, or a predicate symbol applied to terms, $p(t_1, \dots, t_n)$, respecting the arity and sorts of p . A literal is either an atom or its negation. Cubes are conjunctions of literals and clauses are disjunctions of literals. A formula is any Boolean combination of literals, possibly with quantifiers. For a formula φ with free variables, we use $\varphi^\exists$ to denote the existential closure of φ . Unless otherwise stated, we only consider *closed* formulas (i.e., formulas without any free variables). As usual, we use sets of formulas and their conjunctions interchangeably. We use the special term $ite(c, t_1, t_2)$, where c is a formula, and t_1, t_2 are terms of the same sort, to denote an “if then else” operation. We say that a formula θ is *pure* with respect to a function symbol f , if f only appears in positive literals of the form $f(\bar{t}) = x$ where f does not appear in $\bar{t}$, and x is a variable. We call such formulas *f-pure* to emphasize f . Note that every quantifier-free formula θ can be converted into an equivalent *f-pure* formula by introducing existentially quantified variables that represent the f applications. For example, $p(f(f(x)))$ may be rewritten into $\exists y, z \cdot f(x) = y \wedge f(y) = z \wedge p(z)$.

A model M consists of a nonempty domain for each sort, and a valuation function that maps each function and predicate symbol to its interpretation. For a closed formula φ , $M \models \varphi$ is defined in the usual way (see for example chapter 2 of [35]). By convention, we use non bold symbols (like f) to refer to syntactic entities and bold symbols (like $\mathbf{f}$) to refer to their semantics i.e., their valuation in a model. Given a model M over a signature Σ that may or may not include a , we use the notation $M[a \mapsto \mathbf{a}]$ to mean the model over $\Sigma \cup \{a\}$ obtained from M when a is interpreted to $\mathbf{a}$.

1.2 Theories

Throughout the thesis we consider *satisfiability modulo theories*. That is, for all signatures, we assume an underlying theory $\mathcal{T}$ that interprets some sorts, function and predicate symbols. A symbol interpreted by the theory is called an interpreted symbol. We refer to interpreted constants as values. Theory $\mathcal{T}$ may either be defined by a set of axioms (in which case the theory consists of the deductive closure of the axioms), or by a set of intended models (in which case the theory consists of all formulas that are true in these models). We say that a formula φ is *satisfiable* modulo theory $\mathcal{T}$ if there is a *model* M of $\mathcal{T}$ that satisfies the formula, denoted $M \models_{\mathcal{T}} \varphi$. Otherwise, the formula is *unsatisfiable*. φ is *valid* modulo $\mathcal{T}$ if every model of $\mathcal{T}$ satisfies φ . We say that two formulas, θ_1 and θ_2 , are *equisatisfiable* if it holds that there exists $M \models_{\mathcal{T}} \theta_1$ if and only if there exists $M' \models_{\mathcal{T}} \theta_2$. We omit $\mathcal{T}$ and write $M \models \varphi$ when the theory is clear from context.

1.2.1 Equality and Uninterpreted Functions

The theory of Equality and Uninterpreted Functions (EUF) has a signature with the equality symbol (denoted $\approx$), disequality symbol (denoted $\not\approx$), and a set of (uninterpreted) function and predicate symbols. The theory only has a few basic axioms including function congruence, symmetry, and transitivity.

Common decision procedures for EUF revolve around *egraphs*: a data structure to compactly represent a set of terms and an equivalence relation on those terms [88]. We assume that graphs have an ordered successor relation and use $n[i]$ to denote the i th successor (child) of a node n . An out-degree of a node n , $\text{deg}(n)$, is the number of edges leaving n .

Given a node n , $\text{parents}(n)$ denotes the set of nodes with an outgoing edge to n and $\text{children}(n)$ denotes the set of nodes with an incoming edge from n .

$$\varphi_{1.1}(x, y, z) \triangleq z \approx \text{read}(a, x) \wedge k + 1 \approx \text{read}(a, y) \wedge x \approx y \wedge 3 > z$$

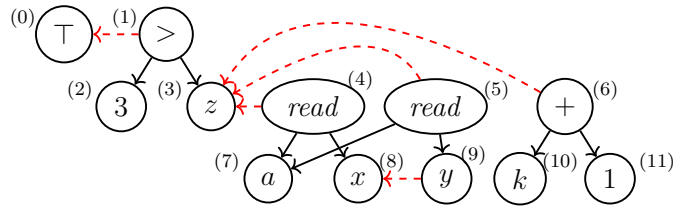


Figure 1.1: Example egraph of $\varphi_{1.1}$.

Definition 1 Let Σ be a first-order logic signature. An *egraph* is a tuple $G = \langle N, E, L, \text{root} \rangle$, where

- (a) $\langle N, E \rangle$ is a directed acyclic graph,
- (b) L maps nodes to function symbols in Σ or logical variables, and
- (c) $\text{root} : N \mapsto N$ maps a node to its *root* such that the relation $\rho_{\text{root}} \triangleq \{(n, n') \mid \text{root}(n) = \text{root}(n')\}$ is an equivalence relation on N that is *closed under congruence*: $(n, n') \in \rho_{\text{root}}$ whenever n and n' are *congruent* under root , i.e., whenever $L(n) = L(n')$, $\text{deg}(n) = \text{deg}(n') > 0$, and, $\forall 1 \leq i \leq \text{deg}(n) \cdot (n[i], n'[i]) \in \rho_{\text{root}}$. $\square$

Given an egraph G , the *class* of a node $n \in G$, $\text{class}(n) \triangleq \rho_{\text{root}}(n)$, is the set of all nodes that are equivalent to n . The term of n , $\text{term}(n)$, with $L(n) = f$ is f if $\text{deg}(n) = 0$ and $f(\text{term}(n[1]), \dots, \text{term}(n[\text{deg}(n)]))$, otherwise. We assume that the terms of different nodes are different, and refer to a node n by its term.

An example of an egraph $G = \langle N, E, L, \text{root} \rangle$ is shown in Fig. 1.1. A symbol f inside a circle depicts a node n with label $L(n) = f$, solid black and dashed red arrows depict E and root , respectively. The order of the black arrows from left to right defines the order of the children. In our examples, we refer to a specific node i by its number using $\mathbf{N}(i)$ or its term, e.g., $\mathbf{N}(k + 1)$. A node n without an outgoing red arrow is its own root. A set of nodes connected to the same node with red edges forms an equivalence class. In this example, root defines the equivalence classes $\{\mathbf{N}(3), \mathbf{N}(4), \mathbf{N}(5), \mathbf{N}(6)\}$, $\{\mathbf{N}(8), \mathbf{N}(9)\}$, and a class for each of the remaining nodes. Examples of some terms in G are $\text{term}(\mathbf{N}(9)) = y$ and $\text{term}(\mathbf{N}(5)) = \text{read}(a, y)$.

1.2.2 Linear Integer Arithmetic

In the theory of Linear Integer Arithmetic (LIA), formulas are defined over a signature that includes interpreted function symbols $+$, $-$, $\times$, interpreted predicate symbols

$<, \leq, |$, interpreted constant symbols $0, 1, 2, \dots$, and uninterpreted constant symbols $a, b, \dots, x, y, \dots$. We write $\mathbb{Z}$ for the set interpreted constant symbols, and call them *integers*. We use *constants* to refer exclusively to the uninterpreted constants (these are often called *variables* in LIA literature). Terms (and accordingly formulas) in LIA are restricted to be *linear*, that is, multiplication is never applied to two constants.

We write $\text{LIA}^{-\text{div}}$ for the fragment of LIA that excludes divisibility ($d \mid h$) predicates. A literal in $\text{LIA}^{-\text{div}}$ is a linear inequality; a cube is a conjunction of such inequalities, that is, a polytope. We find it convenient to use matrix-based notation for representing cubes in $\text{LIA}^{-\text{div}}$. A ground cube $c \in \text{LIA}^{-\text{div}}$ with p inequalities (literals) over k (uninterpreted) constants is written as $A \cdot \bar{x} \leq \bar{n}$, where A is a $p \times k$ matrix of coefficients in $\mathbb{Z}^{p \times k}$, $\bar{x} = (x_1 \cdots x_k)^T$ is a column vector that consists of the (uninterpreted) constants, and $\bar{n} = (n_1 \cdots n_p)^T$ is a column vector in $\mathbb{Z}^p$. For example, the cube $x \geq 2 \wedge 2x + y \leq 3$ is written as $\begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} -2 \\ 3 \end{bmatrix}$. All vectors are column vectors, super-script T denotes transpose, dot is used for a dot product and $[\bar{n}_1; \bar{n}_2]$ stands for a matrix of column vectors $\bar{n}_1$ and $\bar{n}_2$.

1.2.3 Algebraic Data Types

The theory of Algebraic Data Types (ADT) is defined over a signature that contains a sort τ for an algebraic datatype (for simplicity, we assume only one such sort), function symbols for *constructors* and *selectors*, and predicate symbols for *testers*. A constructor for sort τ , denoted C , returns an element of sort τ (its arguments may be of any sort, including τ). A selector for the i th argument of C , denoted S_C^i , receives an element of sort τ and returns an element of the sort of the i th argument of C . A tester for C , denoted IS_C , is defined over sort τ . The theory of ADTs is defined by an *initial model* of the following axiom schema, ξ :

$$\forall \bar{x}. IS_C(C(\bar{x})) \quad \forall \bar{x}. \neg IS_{C'}(C(\bar{x})) \quad \forall \bar{x}. S_C^i(C(\bar{x})) = x_i$$

where $\bar{x} = \{x_1, \dots, x_n\}$ and $C \neq C'$. An initial model is a model where the domain of sort τ consists of all *constructor terms*: these are ground terms where the only ADT function applications are of constructors. The theory of ADT consists of the set of formulas that hold in an initial model of ξ . The theory of ADTs is decidable (even with quantifiers) [105].

Note that an initial model cannot be specified in First Order Logic. However, many SMT solvers support reasoning over it. Accordingly, in the thesis, all models we consider are isomorphic to an initial model. In particular, we write $M \models \theta$ to mean that M is isomorphic to an initial model that satisfies θ .

The List ADT For convenience of presentation, throughout the thesis, we use a combination of the List ADT and Linear Integer Arithmetic, i.e., Lists of Integers, denoting them simply as Lists. The List datatype is defined using a signature that consists of sort s for the elements stored in lists, which is in our case sort Int , and sort List_s for lists. It has two constructors:

- *nil* is a nullary constructor for constructing an empty list, and
- *insert* receives an element of sort s and a list and constructs a list where the element is added to the head of the list.

The *insert* constructor is associated with two selectors: *head* for the first argument, and *tail* for the second argument.

1.2.4 Recursively Defined Functions

Recursive functions are in general not expressible in First Order Logic. In SMT they can be specified by *function axioms*, which can precisely capture the definition of the recursion, but relax the least-fixpoint semantics (e.g., [78]). A function axiom is a formula of the form $\forall \bar{x} \cdot f(\bar{x}) \approx t(\bar{x})$ where t is a term of an appropriate sort. t can contain f itself as well as other function (or constant) symbols. We call a function symbol f for which a function axiom is given, a recursively defined function (RDF), to distinguish it from other uninterpreted function symbols. However, there is really no easy way to tell when a function axiom *defines* a function. Some function axioms are unsatisfiable, for example, $\forall x \cdot f(x) \approx \neg f(x)$. Some axioms provide a complete definition of a function, e.g., $\forall x \cdot f(x) \approx \text{ite}(x \leq 0, 0, x \times f(x - 1))$, which has a unique interpretation over the standard model of arithmetic, and some do not, e.g., $\forall x \cdot f(x) \approx f(x + 1) - 1$, which is satisfied by the interpretation $\mathbf{f}(x) = x$, but also by $\mathbf{f}(x) = x + 1$, $\mathbf{f}(x) = x + 2$, etc. While SMT-LIB does not restrict function axioms to uniquely define a function, this thesis addresses RDFs that are uniquely defined by their axioms. To avoid the need to characterize such functions, we focus on the special case of catamorphisms.

Catamorphisms *Catamorphisms* are a notable example of function axioms that specify well defined functions over ADTs. A catamorphism (or, generalized fold) is an RDF that abstracts the content of an ADT into a *single* value. Common abstractions, such as length of a list, height of a tree, set of all elements of a tree, are all examples of catamorphisms. Given an interpreted sort r for the range of the abstraction, a catamorphism from the list ADT List_s to r is specified using a base case value, b , of sort r , and a term, denoted $y \oplus z$, of sort r , where y and z are variables of sorts s and

r , respectively, and all symbols are interpreted. Intuitively, the base case value b is used to define the value of the catamorphism for nil , while $y \oplus z$ is used to define the value in the recursive case of a list x created by the *insert* constructor, in which case the value is defined by applying $\oplus$ on the element $y = head(x)$ inserted to the list and on the value z recursively defined for $tail(x)$. Formally, given $b, \oplus$, the following function axiom defines f as a catamorphism over $List_s$:

$$\forall x \cdot f(x) = ite(x \approx nil, b, head(x) \oplus f(tail(x)))$$

We denote such a function axiom $\varphi_{(b, \oplus)}$ and call it a catamorphism axiom for f .

Example 1 The *length* of a list is a catamorphism from $List$ to Int specified by:

$$\varphi_{(0, +1)} \triangleq \forall x \cdot length(x) \approx ite(x \approx nil, 0, 1 + length(tail(x))) \quad \square$$

The theory of ADTs with catamorphisms (and, more generally, RDFs) is undecidable.

1.3 Quantifier elimination

Given a quantifier-free (QF) formula φ with free variables $\bar{v}$, *quantifier elimination* (qelim) of $\varphi^\exists$ is the problem of finding a QF formula ψ with no free variables such that $\psi \equiv \varphi^\exists$.

For example, a qelim of $\exists a \cdot (a \approx x \wedge f(a) > 3)$ is $f(x) > 3$; and, there is no qelim of $\exists x \cdot (f(x) > 3)$, because it is impossible to restrict f to have “at least one value in its range that is greater than 3” without a quantifier.

For many applications, qelim is prohibitively expensive or undesirable. In these applications, we under-approximate qelim using *Model Based Projection* (MBP). Given a formula φ with free variables $\bar{v}$, and a model $M \models \varphi$, an MBP of φ relative to M is a QF formula ψ such that $\psi \Rightarrow \varphi^\exists$ and $M \models \psi$. That is, ψ has no free variables, is an under-approximation of φ , and satisfies the designated model M , just like φ .

MBP can also be defined for a formula without free variables. Given a list of constants $\bar{v}$, a formula φ and a model $M \models \varphi$, $MBP(\bar{v}, \varphi, M)$ is a formula over $\Sigma \setminus \bar{v}$ such that $M \models MBP(\bar{v}, \varphi, M)$ and any model $M' \models MBP(\bar{v}, \varphi, M)$ can be extended to a model $M'' \models \varphi$ by providing an interpretation for $\bar{v}$. There are polynomial time algorithms for computing MBP for many theories including Linear Arithmetic [15], [71], arrays [70], and ADTs [15]. The following example illustrates MBP:

Example 2 Consider the formula φ : $a' \approx a + 1 \wedge b' \approx b + 1 \wedge a' \leq 5 \wedge b' > 5$, and its model $M_1 = [a \mapsto 4, b \mapsto 5, a' \mapsto 5, b' \mapsto 6]$. Possible MBPs of φ relative to M_1 include:

$$\psi_1 \equiv a \leq 4 \wedge b > 4$$

$$\psi_2 \equiv a < b \wedge b = 5$$

Note that ψ_1 is equivalent to, and ψ_2 is a proper under-approximation of $\exists a', b' \cdot \varphi$, respectively. The soundness of our algorithms do not depend on the specific choice of MBP. $\square$

1.4 Interpolation

Given an unsatisfiable formula $A \wedge B$, an *interpolant* [26], denoted $\text{ITP}(A, B)$, is a formula I over the shared signature of A and B such that $A \Rightarrow I$ and $I \Rightarrow \neg B$. In model checking, interpolation is widely used as a generalization technique: when A is the set of reachable states of a system and B is the set of Bad state, the interpolant is a candidate inductive invariant that approximates reachable states and contradicts Bad states (see for example [1], [2], [16], [80], [101]). In theories like Linear Arithmetic, interpolation techniques (for example those based on Farkas Lemma) introduce new terms that are quite effective in blocking Bad states.

Example 3 Consider the formula $A: a \approx 0 \wedge b \approx 0 \wedge a' \approx a + 1 \wedge b' \approx b + 1$ and $B: a' < b' \wedge b' \approx 5$. Clearly, $A \wedge B$ is unsatisfiable. Two possible interpolants for $A \wedge B$ are $b' < 2$ and $a' \approx b'$. Both interpolants contain terms not present in the original formula. $\square$

1.5 Transition Systems

A *transition system* (TS) is a pair $\langle \text{Init}, \text{Tr} \rangle$, where Init is a formula over some signature Σ and Tr is a formula over $\Sigma \cup \Sigma'$, where $\Sigma' = \{s' \mid s \in \Sigma\}$.¹ The states of the system correspond to structures over Σ , Init represents the initial states and Tr represents the transition relation, where Σ is used to represent the pre-state of a transition, and Σ' is used to represent the post-state. For a formula φ over Σ , we denote by φ' the formula obtained by substituting each $s \in \Sigma$ by $s' \in \Sigma'$. A *safety problem* is a triple $\langle \text{Init}, \text{Tr}, \text{Bad} \rangle$, where $\langle \text{Init}, \text{Tr} \rangle$ is a transition system and Bad is a formula over Σ representing a set of bad states. We say that a transition system is *unsafe* with respect to a set of *Bad* states if there is a sequence of states $s_0, s_1, \dots, s_n$, called a counterexample, such that $s_0 \in \text{Init}$, for all i , $(s_i, s'_{i+1}) \in \text{Tr}$, and $s_n \in \text{Bad}$. A system is *safe* otherwise. The existence of inductive invariants establish the safety of a system. An inductive invariant is a formula Inv over Σ such that $\text{Init} \Rightarrow \text{Inv}$, $\text{Inv} \wedge \text{Tr} \Rightarrow \text{Inv}'$, and $\text{Inv} \Rightarrow \neg \text{Bad}$. Many algorithms for the safety problem attempt to prove the system safe by constructing an inductive invariant.

¹In fact, a primed copy is introduced in Σ' only for the uninterpreted symbols in Σ . Interpreted symbols remain the same in Σ' .

1.6 IC3/PDR

IC3 [17] is an algorithm for solving the safety problem for boolean transition systems. PDR [33] is a reimplementation of the key ideas in IC3. They have inspired several finite state and infinite state unbounded model checking algorithms [22], [54], [60], [101] including SPACER. In this section, we give a brief overview of the IC3 algorithm.

The main datastructure is a sequence of formulas $F_0, F_1, \dots$ called *trace*. The formula in a trace, called *frames*, are over state variables $\bar{v}$ (uninterpreted constants in Σ) and are in conjunctive normal form (CNF). The clauses in a frame are called *lemmas*. IC3 maintains the invariant that frame F_i overapproximates states reachable in i steps of the transition system and are monotonic: $Init \Rightarrow F_i$, $F_i \wedge Tr \Rightarrow F_{i+1}$, $F_i \Rightarrow F_{i+1}$. Frame F_i is *safe* if it excludes bad states: $F_i \Rightarrow \neg Bad$. Since frames are monotonic, safety of one frame implies safety of all frames before it. IC3 iteratively establishes safety of higher and higher frames. To establish the safety of a frame, it creates predecessors to *Bad* states and blocks them to learn new lemmas. A predecessor to a set of states $\psi(\bar{v})$ is a set of states $\varphi(\bar{v})$ s.t. $\varphi \Rightarrow \exists \bar{v}' \cdot Tr \wedge \psi'$. A *proof obligation* (POBs) is a tuple $\langle \varphi, i \rangle$ of a predecessor states φ and a number i s.t. if any state in φ is reachable in i steps, *Bad* is reachable from *Init* in $i + 1$ or more steps. IC3 maintains a queue Q of POBs.

IC3 initializes the trace by assigning the first frame as *Init* and all other frames as $\top$. IC3 initializes the POB queue by adding $\langle Bad, 1 \rangle$ to it. In each iteration, IC3 queries Q for a POB $\langle \varphi, i + 1 \rangle$ and attempts to *block* it. The POB is blocked by frame F_i if $F_i \wedge Tr \wedge \varphi'$ is unsatisfiable. If so, IC3 learns a new lemma by generalizing $\neg \varphi$. IC3 employs a generalization strategy called inductive generalization. Inductive generalization starts with a lemma $\ell = \neg \varphi$ and drops literals from ℓ as long as the following holds: $((\ell \wedge F_i \wedge Tr) \vee Init') \Rightarrow \ell'$. Intuitively, inductive generalization finds a smaller (more general) lemma as long as it is inductive *relative* to the frame F_i . Once generalization is done, IC3 adds lemma ℓ to all frames including and before F_{i+1} . If blocking is unsuccessful (i.e. $F_i \wedge Tr \wedge \varphi'$ is satisfiable), IC3 computes a predecessor POB for φ by projecting $F_i \wedge Tr \wedge \varphi'$ onto the $\bar{v}$ variables. IC3 employs a simple strategy of getting a full satisfying assignment for $F_i \wedge Tr \wedge \varphi'$ and collecting the assignments to $\bar{v}$ variables. PDR goes one step further and generalizes predecessor POB by dropping assignments that do not affect the satisfiability (called *ternary simulation*). Once created, the predecessor POB $\langle \psi, i \rangle$ is added to the proof obligation queue. This way, in each iteration, IC3 either creates a new predecessor POB or learns a new lemma. The process continues until a POB $\langle \varphi, 0 \rangle$ intersects with *Init* (i.e. $Init \wedge \varphi$ is satisfiable), in which case we have a counterexample, or 2 consecutive safe frames F_i, F_{i+1} are syntactically equal, in which case F_i is a safe inductive invariant that establishes safety.

1.7 Constrained Horn Clauses

Given a signature Σ , a background theory $\mathcal{T}$ over Σ , and a set P of uninterpreted predicate symbols not in Σ (but defined over the sorts in Σ), a Constrained Horn Clause (CHC) is a closed formula of the form

$$\forall \bar{x}, \bar{x}' \cdot \phi(\bar{x}, \bar{x}') \wedge \bigwedge_i p_i(\bar{x}_{p_i}) \Rightarrow h(\bar{x}'_h) \quad (1.1)$$

where $p_i \in P$, $h \in P \cup \{\perp\}$, $\bar{x}' = \{x' \mid x \in \bar{x}\}$, ϕ is a quantifier-free formula over Σ with free variables in $\bar{x} \cup \bar{x}'$, $\bar{x}_{p_i} \subseteq \bar{x}$, and $\bar{x}'_h \subseteq \bar{x}'$. The same unknown predicate $p \in P$ can appear multiple times in a CHC with different arguments (i.e., in the equation above p_i is not necessary distinct from p_j when $i \neq j$). The left hand side of the implication is called the *body* or *tail* of the CHC, and the right hand side is called its *head*. We refer to ϕ as the *constraint*. A set of CHCs is a conjunction of CHCs. With abuse of notation, we sometimes refer to a set of CHCs as a CHC. A (single) CHC is *linear* if the tail has at most one uninterpreted predicate symbol. A set of CHCs is linear if all the CHCs in it are linear. Otherwise, the set of CHCs is *non-linear*. When a set of CHCs is satisfiable (modulo $\mathcal{T}$), a *solution* is a model, including an interpretation of all the predicates in P , that satisfies all the CHCs. Typically, we are interested in solutions that are expressed as formulas over some class of formulas $\mathcal{A}$ [14]:

Definition 2 ($\mathcal{A}$ -Solutions) Let $\mathcal{A}$ be a class of formulas. An $\mathcal{A}$ -solution for a set of CHCs C assigns to each predicate symbol $p \in P$ a formula $\theta_p(\bar{y}) \in \mathcal{A}$ whose free variables $\bar{y}$ correspond to the arguments of p such that replacing each predicate application $p(\bar{x}_p)$ in C by $\theta_p[\bar{y} \mapsto \bar{x}_p]$ results in a valid closed formula (modulo the background theory). $\square$

It is possible that a set of CHCs is satisfiable, but has no $\mathcal{A}$ -solution.

The safety of a TS can be checked by solving a set of linear CHCs that encode the TS. All linear CHCs are reducible to a set of CHCs with a single unknown predicate *Inv* and exactly 3 clauses (free variables are implicitly universally quantified):

$$Init(\bar{x}) \Rightarrow Inv(\bar{x}) \quad Inv(\bar{x}) \wedge Tr(\bar{x}, \bar{x}') \Rightarrow Inv(\bar{x}') \quad Inv(\bar{x}) \wedge Bad(\bar{x}) \Rightarrow \perp \quad (1.2)$$

A solution for *Inv* is an *inductive invariant* that proves that the TS is *safe*: no bad states are reachable from an initial state via the *Tr*. When the CHC is unsatisfiable, there exists a *counterexample* to safety. Counterexamples to the safety property are derived from resolution proofs of unsatisfiability of CHCs.

Algorithm 1: SPACER algorithm as a set of guarded commands. We use the shorthand $\mathcal{F}(\varphi) = R' \vee (\varphi \wedge Tr)$.

```

function SPACER:
In:  $\langle Init, Tr, Bad \rangle$ 
Out:  $\langle \text{SAFE}, Inv \rangle$  or UNSAFE
 $Q := \emptyset$  // pob queue
 $N := 0$  // maximum safe level
 $\mathcal{O}_0 := Init, \mathcal{O}_i := \top$  for all  $i > 0$  // lemma trace
 $R := Init$  // reachable states
forever do
  Candidate  $\llbracket \text{ISAT}(\mathcal{O}_N \wedge Bad) \rrbracket Q := Q \cup \langle Bad, N \rangle$ 
  Predecessor  $\llbracket \langle \varphi, i+1 \rangle \in Q, M \models \mathcal{O}_i \wedge Tr \wedge \varphi' \rrbracket$ 
   $Q := Q \cup \langle \text{MBP}(\bar{x}', Tr \wedge \varphi', M), i \rangle$ 
  Successor  $\llbracket \langle \varphi, i+1 \rangle \in Q, M \models \mathcal{F}(R) \wedge \varphi' \rrbracket$ 
   $R := R \vee \text{MBP}(\bar{x}, \mathcal{F}(R), M)[\bar{x}' \mapsto \bar{x}]$ 
  Conflict  $\llbracket \langle \varphi, i+1 \rangle \in Q, \mathcal{F}(\mathcal{O}_i) \Rightarrow \neg \varphi' \rrbracket$ 
   $\mathcal{O}_j := (\mathcal{O}_j \wedge \text{ITP}(\mathcal{F}(\mathcal{O}_i), \varphi')[\bar{x}' \mapsto \bar{x}])$  for all  $j \leq i+1$ 
  Induction  $\llbracket \ell \in \mathcal{O}_{i+1}, \ell = (\varphi \vee \psi), \mathcal{F}(\varphi \wedge \mathcal{O}_i) \Rightarrow \varphi' \rrbracket$ 
   $\mathcal{O}_j := \mathcal{O}_j \wedge \varphi$  for all  $j \leq i+1$ 
  Propagate  $\llbracket \ell \in \mathcal{O}_i, \mathcal{O}_i \wedge Tr \Rightarrow \ell' \rrbracket \mathcal{O}_{i+1} := (\mathcal{O}_{i+1} \wedge \ell)$ 
  Unfold  $\llbracket \mathcal{O}_N \Rightarrow \neg Bad \rrbracket N := N + 1$ 
  Safe  $\llbracket \mathcal{O}_{i+1} \Rightarrow \mathcal{O}_i$  for some  $i < N \rrbracket$  return  $\langle \text{SAFE}, \mathcal{O}_i \rangle$ 
  Unsafe  $\llbracket \text{ISAT}(Bad \wedge R) \rrbracket$  return UNSAFE

```

1.8 Spacer

SPACER [71] is a semi-decision procedure for CHC satisfiability implemented in the Z3 SMT-solver [85]. Just like IC3/PDR, SPACER computes predecessors to *bad* states and blocks them and generalizes to learn lemmas until either a counter example or a solution is found. In the rest of this chapter, we explain SPACER in detail. We first describe SPACER for the simple case of linear CHCs encoding TS and then explain the full algorithm for general, non-linear CHCs.

1.8.1 Spacer for linear CHCs

As mentioned earlier, Linear CHCs are reducible to a set of 3 CHCs with a single uninterpreted predicate (Eq. (1.2)). We denote such CHCs using the three tuple $\langle Init, Tr, Bad \rangle$. Without loss of generality, we can assume that *Bad* is a cube.

Algorithm 2: The SPACER algorithm for linear CHCs.

In: $\langle \text{Init}, Tr, \text{Bad} \rangle$ Out: An Inductive invariant or UNSAFE 1 function GSpacer: /* Initialize state of the solver */ 2 $Q := \emptyset; N := 0; R := \text{Init};$ 3 $\mathcal{O}_0 := \text{Init}; \mathcal{O}_i := \top, \forall i > 0$ 4 Enqueue($Q, \langle \text{Bad}, 0 \rangle$) 5 while $\top$ do 6 $\langle \varphi, i \rangle := \text{Pop}(Q)$ 7 if $\text{ISAT}(\mathcal{F}(\mathcal{O}_{i-1}) \wedge \varphi')$ then // The pob φ cannot be blocked at i 8 AddPredecessor($\langle \varphi, i \rangle$) 9 if $\text{ISAT}(R \wedge \text{Bad})$ then return UNSAFE // Unsafe 10 else // The pob φ can be blocked at i 11 Block($\langle \varphi, i \rangle$) 12 for $0 \leq j \leq N$ do 13 for $\ell \in \mathcal{O}_j \setminus \mathcal{O}_{j+1}$ do 14 if $\mathcal{O}_j \wedge Tr \Rightarrow \ell'$ then $\mathcal{O}_{j+1} := \mathcal{O}_{j+1} \wedge \ell$ // Propagate 16 if $\exists 0 \leq j < N \cdot \mathcal{O}_j \Rightarrow \mathcal{O}_{j-1}$ then 17 return $\langle \text{SAFE}, \mathcal{O}_j \rangle$ // Safe 18 if $\mathcal{O}_N \Rightarrow \neg \text{Bad}$ then $N := N + 1$ // Unfold 20 Push($Q, \langle \text{Bad}, N \rangle$)	21 function AddPredecessor: 23 if $\text{ISAT}(\mathcal{F}(R) \wedge \varphi')$ then 25 find M_1 s.t. $M_1 \models \mathcal{F}(R) \wedge \varphi'$ 27 $s := (\text{MBP}(\bar{x}, \mathcal{F}(R), M_1)[\bar{x}' \mapsto \bar{x}])$ 29 $R := R \vee s$ // Successor 31 return 33 find M_2 s.t. $M_2 \models \Theta$ 35 $p := \text{MBP}(\bar{x}', Tr \wedge \varphi', M_2)$ 37 Push($Q, \langle p, i - 1 \rangle$) // Predecessor 39 Push($(Q, \langle \varphi, i \rangle)$) 40 function Block: 42 $\ell_1 := \text{ITP}(\mathcal{F}(\mathcal{O}_{i-1}), \varphi')$ // Conflict 43 $\ell_2 := \alpha$ s.t. $\ell_1 = \alpha \vee \beta$ and $\mathcal{F}(\mathcal{O}_{i-1} \wedge \alpha) \Rightarrow \alpha'$ // Induction 44 for $0 \leq j \leq i$ do $\mathcal{O}_j := \mathcal{O}_j \wedge \ell_2$
---	---

Alg. 1 presents the key ingredients of SPACER as a set of guarded commands (or rules). It maintains the following. Current unrolling depth N at which a counterexample is searched (there are no counterexamples with depth less than N). A *trace* $\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \dots)$ of *frames*, such that each frame $\mathcal{O}_i$ is a set of *lemmas*, and each lemma $\ell \in \mathcal{O}_i$ is a clause. A queue of POBs Q , where each proof obligation in Q is a pair $\langle \varphi, i \rangle$ of a cube φ and a level number i , $0 \leq i \leq N$. An under-approximation R of reachable states. Intuitively, each frame $\mathcal{O}_i$ is a candidate inductive invariant s.t. $\mathcal{O}_i$ over-approximates states reachable up to i steps from Init . The latter is ensured since $\mathcal{O}_0 = \text{Init}$, the trace is monotone, i.e., $\mathcal{O}_{i+1} \subseteq \mathcal{O}_i$, and each frame is inductive *relative* to its previous one, i.e., $\mathcal{O}_i \wedge Tr \Rightarrow \mathcal{O}'_{i+1}$. Each POB $\langle \varphi, i \rangle$ in Q corresponds to a suffix of a potential counterexample that has to be blocked in $\mathcal{O}_i$, i.e., has to be proven unreachable in i steps.

The **Candidate** rule adds an initial POB $\langle \text{Bad}, N \rangle$ to the queue. If a POB $\langle \varphi, i \rangle$ cannot be blocked because φ is reachable from frame $(i - 1)$, the **Predecessor** rule

```

1  a, b := 0, 0;
2  while(nd())
3  // inv: a = b;
4  {
5    a++;
6    b++;
7  }
8  assert(a ≤ 5 ⇒ b ≤ 5);

```

Figure 1.2: An example of a simple program.

generates a predecessor ψ of φ using MBP and adds $\langle \psi, i - 1 \rangle$ to Q . The **Successor** rule updates the set of reachable states if the POB is reachable. If the POB is blocked, the **Conflict** rule strengthens the trace $\mathcal{O}$ by using interpolation to learn a new lemma ℓ that blocks the POB, i.e., ℓ implies $\neg\varphi$. The **Induction** rule strengthens a lemma by inductive generalization and the **Propagate** rule pushes a lemma to a higher frame. If the *Bad* state has been blocked at N , the **Unfold** rule increments the depth of unrolling N . In practice, the rules are scheduled to ensure progress towards finding a counterexample.

To make the algorithm more concrete, Alg. 2 presents an instantiation of these rules. The comments to the right side of a line refer to the abstract rules in Alg. 1. SPACER iteratively computes predecessors (line 9) and blocks them (line 13) in an infinite loop. Whenever a POB is proven to be reachable, the reachable states are updated (line 29). If *Bad* intersects with a reachable state, SPACER terminates and returns UNSAFE (line 9). If one of the frames is an inductive invariant, SPACER terminates with SAFE (line 17). Once a POB $\langle \varphi, i \rangle$ is blocked, SPACER uses interpolation (lines 42) to learn a lemma that block φ and other POBs that are similar to it.

We illustrate SPACER using the following example.

Example 4 As an illustrative example, we will explain how SPACER (Alg. 2) checks the safety of the program in Fig. 1.2. For the program in Fig. 1.2, *Init* is $a \approx 0 \wedge b \approx 0$, *Tr* is $a' \approx a + 1 \wedge b' \approx b + 1$, and *Bad* states are $(a \leq 5 \wedge b > 5)$. Blocking *Bad* at levels 0 and 1 is straightforward. To block *Bad* at level 2, SPACER computes a predecessor using MBP. Let us assume MBP produces the POB $(a < b \wedge b \approx 5)$ (see Ex. 2). SPACER then checks whether the newly created POB $(a < b \wedge b \approx 5)$ is reachable at level 1. That is, SPACER checks $\text{ISAT}(\text{Init} \wedge \text{Tr} \wedge a' < b' \wedge b' \approx 5)$ (line 7). Since this is unsatisfiable, SPACER employs interpolation to learn a lemma. As we saw in Ex. 3, there are at least 2 possibilities for the interpolant.

If SPACER learns the lemma $a \approx b$, SPACER not only blocks the POB $(a < b \wedge b \approx 5)$ at level 1, it also *propagates* the lemma $a \approx b$ (line 15) and detects that it is inductive.

Since $a \approx b$ is also enough to block Bad , SPACER returns SAFE (line 17).

On the other hand, if SPACER learns the lemma $b < 2$, SPACER blocks the POB ($a < b \wedge b \approx 5$) at level 1 but not at any higher levels. The lemma is not strong enough to block Bad at level 2. Therefore, in the next iteration, SPACER computes more predecessors of Bad at level 1 and attempts to block them.

Clearly, the lemmas that SPACER learns determine how fast it can prove the safety of the program. However, it is interpolation that decides which lemma SPACER learns. And interpolation is guided by the (local) heuristics inside the SMT solver. $\square$

1.8.2 Spacer for linear and non-linear CHCs

In this section, we explain the full SPACER algorithm that works on both linear and non-linear CHCs. Note that, conceptually, there is no difference between the SPACER algorithm (Alg. 2) for linear CHCs and the SPACER algorithm for linear and non-linear CHCs (Alg. 3). The only difference is in presentation. We believe it is easier to understand SPACER when we first explain it for CHCs with just one uninterpreted predicate and then lift it to the case when there are multiple.

Alg. 3 explains SPACER when the input is linear or non-linear CHCs. The input SPACER are CHCs over a set of uninterpreted predicates P . Without loss of generality, we assume that there is only one query clause. That is, the input to the algorithm is of the form:

$$\bigwedge_j \left(\forall \bar{x}, \bar{x}' \cdot Tr_j(\bar{x}, \bar{x}') \wedge \bigwedge_{\mathcal{P} \in \overline{UT}(j)} \mathcal{P}(\bar{x}_{\mathcal{P}}) \Rightarrow h_j(\bar{x}'_{h_j}) \right) \wedge \quad (1.3)$$

$$\forall \bar{x}, \bar{x}' \cdot Tr_{Bad}(\bar{x}, \bar{x}') \wedge \bigwedge_{\mathcal{P} \in \overline{UT}^{Bad}} \mathcal{P}(\bar{x}_{\mathcal{P}}) \wedge Bad(\bar{x}'_{Bad}) \Rightarrow \perp$$

Where Tr_j , $\overline{UT}(j)$ and h_j are the constraint, set of uninterpreted predicates, and the head predicate in the j th horn clause. Similarly, Tr_{Bad} and $\overline{UT}^{Bad}$ are the constraint and the set of uninterpreted predicates in the query clause.

Notation All formulas in the algorithm are over state variables $\bar{x}$. For a formula θ over state variables, we use the shorthand $\theta[\bar{y}]$ to mean $\theta[\bar{x} \mapsto \bar{y}]$. We assume that whenever a predicate $\mathcal{P}$ appears in the head of a Horn clause, it has the same variables as its arguments. We use the notation $\overline{UT}^{\mathcal{P}}$ to represent the *uninterpreted tail* of predicate $\mathcal{P}$: the sequence of all pairs $\langle \mathcal{Q}, \bar{v} \rangle$ where $\mathcal{Q}$ is an uninterpreted predicate and the literal $\mathcal{Q}(\bar{v})$ appears in the tail of a Horn clause whose head is $\mathcal{P}$. We use the

Algorithm 3: The SPACER algorithm for linear and non-linear CHCs.

Input: A set of CHCs of the form Eq. (1.3)
over uninterpreted predicates P

Output: SAT/UNSAT

```

1  $R^P := \text{Init}^P \forall \mathcal{P} \in P$ ;
2  $\bar{O}_0^P := \text{Init}^P = \perp? \top : \text{Init}^P \forall \mathcal{P} \in P$ ;
3  $\bar{O}_i^P := \top \forall \mathcal{P} \in P, i > 0$ ;
4  $N := 0; k := 0$ 
5 create_pred_bad()
6 while  $\top$  do
7    $\langle \theta, i+1, \mathcal{P} \rangle := Q.\text{pop}()$ 
8    $\mathcal{I}, M, \text{res} := \text{check\_reach}(\langle \theta, i, \mathcal{P} \rangle)$ 
9   if  $\text{res}$  then
10     create_pred( $\mathcal{I}, \langle \theta, i+1, \mathcal{P} \rangle, M$ )
11     if is_bad_rch() then return UNSAFE
12   else
13     block-pob( $\langle \theta, i+1, \mathcal{P} \rangle$ )
14      $Q := Q \cup \langle \theta, i+2, \mathcal{P} \rangle$ 
15     if is_bad_blkd() then
16        $N := N + 1$ 
17       create_pred_bad()
18     if chk_ind() then return SAFE
19 function chk_ind():
20   return
21      $\exists 0 \leq j \leq N-1 \cdot (\bigwedge_{\mathcal{P} \in P} \bar{O}_{j+1}^P \Rightarrow (\bigwedge_{\mathcal{P} \in P} \bar{O}_j^P))$ 
22 function check_reach( $\langle \theta, i, \mathcal{P} \rangle$ ):
23    $\beta := \mathcal{F}_{\mathcal{P}}(\bar{O}_i) \wedge \varphi'$ 
24   if  $\neg \text{ISAT}(\beta)$  then return  $\neg, \perp$ 
25    $\mu := \{R^Q[\bar{v}] \mid \langle Q, \bar{v} \rangle \in \overline{UT}^{\mathcal{P}}\}$ 
26    $M := \text{max\_sat}(\beta, \mu)$ 
27    $\mathcal{I} := \text{Implicant}(\beta, M)$ 
28    $\mathcal{I} := \mathcal{I} \wedge \{r \mid r \in \mu \wedge M \models r\}$ 
29   return  $\mathcal{I}, M, \top$ 
29 function is_bad_rch():
30    $\theta := \bigwedge_{\mathcal{P} \in \overline{UT}(Bad)} (R^P) \wedge Tr_{Bad} \wedge Bad$ 
31   return  $\text{ISAT}(\theta)$ 
32 function block_pob( $\langle \theta, i+1, \mathcal{P} \rangle$ ):
33    $\beta := \mathcal{F}_{\mathcal{P}}(\bar{O}_i)$ 
34    $c := \text{ITP}(\beta, \theta')[\bar{x}' \mapsto \bar{x}]$ 
35    $\ell := \text{IND\_GEN}(c)$ 
36    $\bar{O}_j^P := \bar{O}_j^P \wedge \ell \quad \forall j \leq i+1$ 
37 function create_pred_bad():
38    $\theta := \bigwedge_{\mathcal{P} \in \overline{UT}(Bad)} \bar{O}_{N+1}^P \wedge Tr_{Bad} \wedge Bad$ 
39   if  $\neg \text{ISAT}(\theta)$  then return
40   Let  $M \models \theta; \bar{v} := \text{vars}(\theta)$ 
41   for  $\langle Q, \bar{y} \rangle \in \overline{UT}^{Bad}$  do
42      $pob := \text{MBP}(\bar{v} \setminus \{\bar{y}\}, \theta, M)$ 
43      $Q := Q \cup \langle pob, N+1, Q \rangle$ 
43 function create_pred( $\mathcal{I}, \langle \theta, i+1, \mathcal{P} \rangle, M$ ):
44    $\Phi := \{\langle Q, \bar{u} \rangle \mid \langle Q, \bar{u} \rangle \in \overline{UT}^{\mathcal{P}} \wedge M \not\models R^Q[\bar{u}]\}$ 
45   if  $\Phi = \emptyset$  then
46     // compute successor
47      $\bar{v} := \text{vars}(\mathcal{I}) \setminus \{\bar{x}'\}$ 
48      $r := \text{MBP}(\bar{v}, \mathcal{I} \wedge \theta', M)$ 
49      $R^P := R^P \vee r[\bar{x}' \mapsto \bar{x}]$ 
50   else
51      $kids := \emptyset$ 
52     for  $\langle Q, \bar{u} \rangle \in \Phi$  do
53        $\bar{v} := \text{vars}(\mathcal{I}) \setminus \{\bar{u}\}$ 
54        $\gamma := \text{MBP}(\bar{v}, \mathcal{I} \wedge \theta', M)$ 
55        $\gamma := \gamma[\bar{u} \mapsto \bar{x}]$ 
56        $kids := kids \cup \langle \gamma, i, Q \rangle$ 
57      $Q := Q \cup kids$ 
58      $Q := Q \cup \langle \theta, (i+1), \mathcal{P} \rangle$ 
58 function is_bad_blkd():
59    $\theta := \bigwedge_{\mathcal{P} \in \overline{UT}(Bad)} \bar{O}_N^P \wedge Tr_{Bad} \wedge Bad$ 
60   return  $\neg \text{ISAT}(\theta)$ 

```

notation $\overline{UT}_i^{\mathcal{P}}$ to mean the uninterpreted tail of the i th Horn clause whose head is $\mathcal{P}$. Let $\mathcal{Y}$ be a map from uninterpreted predicates to their summaries, where a summary is a formula over $\bar{x}$, and $\mathcal{Y}^Q$ is the summary of uninterpreted predicate Q in $\mathcal{Y}$. We

use the shorthand $\mathcal{F}_{\mathcal{P}}(\mathcal{Y})$ to denote the *predicate transformer* of $\mathcal{P}$ defined as:

$$\mathcal{F}_{\mathcal{P}}(\mathcal{Y}) = (Init^{\mathcal{P}})' \vee \bigvee_i \left(\left(\bigwedge_{\langle \mathcal{Q}, \bar{v} \rangle \in \overline{UT}_i^{\mathcal{P}}} \mathcal{Y}^{\mathcal{Q}}[\bar{x} \mapsto \bar{v}] \right) \wedge Tr_i \right)$$

where $Init^{\mathcal{P}}$ is the disjunction of tails of Horn clauses which have no uninterpreted predicates in their tails and whose head is $\mathcal{P}$ and Tr_i is the constraint in the i th Horn clause whose head is $\mathcal{P}$. We use R to denote the map from uninterpreted predicates to their must summaries. Similarly, we use $\mathcal{O}_i$ to denote the map from uninterpreted predicates to their may summaries in the i 'th frame. We denote the sequence $\mathcal{O}_0, \mathcal{O}_1, \dots$ by $\bar{\mathcal{O}}$, and write $\bar{\mathcal{O}}_i$ to refer to $\mathcal{O}_i$. A POB is represented as a triple $\langle \theta, i, \mathcal{P} \rangle$, where θ is formula in the background theory, i is a natural number representing the level at which θ is to be blocked and $\mathcal{P}$ denotes the uninterpreted predicate to which the POB belongs.

SPACER iteratively calls the *check_reach* function to check the reachability of a POB at a particular level (line 8). The *check_reach* function first checks whether the POB is blocked using may summaries (line 23). If the check is satisfiable, the function finds a model that satisfies as many must summaries as possible using a *max_sat* algorithm. The *check_reach* function returns $\top$ only if the may summaries are not strong enough to block the POB at this level. In this case, the algorithm calls the *create_pred* function to create and add predecessor POBs to the queue (line 10). Otherwise, it calls the *block_pob* to learn a lemma that blocks the POB (line 13). The *create_pred* function (line 43) uses MBP to create a predecessor POB for all the predicates in the tail whose must summaries are *not* satisfied by the model (but their may summaries are). If there are no such predicates, the POB is reachable from the must summaries. In this case, the algorithm computes a reachable state that intersects with the POB using MBP (line 47) and adds it to the set of reachable states. SPACER routinely checks whether *Bad* is reachable (line 11) and increases the depth of exploration when *Bad* is blocked (line 15).

Chapter 2

Global Guidance For Linear Arithmetic

SMT-based Model Checking algorithms that combine SMT-based search for bounded counterexamples with interpolation-based search for inductive invariants are currently the most effective techniques for verification of infinite state systems. They are widely applicable, including for verification of synchronous systems, protocols, parameterized systems, and software.

The Achilles heel of these approaches is the mismatch between the *local* reasoning used to establish absence of bounded counterexamples and a *global* reason for absence of unbounded counterexamples (i.e., existence of an inductive invariant). This is particularly apparent in IC3-style algorithms, such as SPACER. IC3-style algorithms establish bounded safety by repeatedly computing predecessors of error (or bad) states, blocking them by local reasoning about a single step of the transition relation of the system, and, later, using the resulting *lemmas* to construct a candidate inductive invariant for the global safety proof. The whole process is driven by the choice of local lemmas. Good lemmas lead to quick convergence, bad lemmas make even simple-looking problems difficult to solve.

The effect of local reasoning is somewhat mitigated by the use of interpolation in lemma construction. In addition to the usual inductive generalization by dropping literals from a blocked bad state, interpolation is used to further generalize the blocked state using theory-aware reasoning. For example, when blocking a bad state $x \approx 1 \wedge y \approx 1$, inductive generalization would infer a sub-clause of $x \not\approx 1 \vee y \not\approx 1$ as a lemma, while interpolation might infer $x \not\approx y$ – a predicate that might be required for the inductive invariant. SPACER, which is based on this idea, is extremely effective, as demonstrated by its performance in CHC-COMP competitions [56]. The downside, however, is that the approach leads to a highly unstable procedure that is extremely

sensitive to syntactic changes in the system description, changes in interpolation algorithms, and any algorithmic changes in the underlying SMT-solver.

An alternative approach, often called *invariant inference*, is to focus on the global safety proof, i.e., an inductive invariant. This has long been advocated by such approaches as Houdini [41], and, more recently, by a variety of machine-learning inspired techniques, e.g., FreqHorn [37], LinearArbitrary [106], and ICE-DT [47]. The key idea is to iteratively generate positive (i.e., reachable states) and negative (i.e., states that reach an error) examples and to compute a candidate invariant that separates these two sets. The reasoning is more focused towards the invariant, and, the search is restricted by either predicates, templates, grammars, or some combination. Invariant inference approaches are particularly good at finding simple inductive invariants. However, they do not generalize well to a wide variety of problems. In practice, they are often used to complement other SMT-based techniques.

In this chapter, we present a novel approach that extends, what we call, *local reasoning* of IC3-style algorithms with *global guidance* inspired by the invariant inference algorithms described above. Our main insight is that the set of lemmas maintained by IC3-style algorithms hint towards a potential global proof. However, these hints are lost in existing approaches. We observe that letting the current set of lemmas, that represent candidate global invariants, guide local reasoning by introducing new lemmas and states to be blocked is often sufficient to direct IC3 towards a better global proof.

We present and implement our results in SPACER (see Sec. 1.8). However, our results are fundamental and apply to any other IC3-style algorithm. While our implementation works with arbitrary CHC instances, we simplify the presentation by focusing on infinite state model checking of transition systems.

We compare the new strategy, called GSPACER, to the original implementation of SPACER. We show that GSPACER outperforms SPACER in benchmarks from CHC-COMP 2018, 2019, 2020, and 2021. More significantly, we show that the performance is independent of interpolation. While SPACER is highly dependent on interpolation parameters, and performs poorly when interpolation is disabled, the results of GSPACER are virtually unaffected by interpolation. We also compare GSPACER to LinearArbitrary [106], a tool that *infers invariants* using global reasoning. GSPACER outperforms LinearArbitrary on the benchmarks from [106]. These results indicate that global guidance mitigates the shortcomings of local reasoning.

The rest of the chapter is structured as follows. Sec. 2.1 illustrates pitfalls of local generalization using three examples. Sec. 2.2 introduces our *global guidance* as a set of abstract inference rules. Sec. 2.3 describes an instantiation of the rules to Linear Integer Arithmetic (LIA) and Linear Rational Arithmetic (LRA). Sec. 2.4 presents our empirical evaluation. Sec. 2.5 describes related work and Sec. 2.6 concludes the

<pre> 1 a, c := 0, 0; 2 // b, d := a, c; 3 b, d := 0, 0; 4 while(nd()) 5 // inv: a - c = b - d; 6 { 7 if(nd()) { a++; b++; } 8 else { c++; d++; } 9 } 10 assert(a ≤ c ⇒ b ≤ d); </pre>	<pre> a, b := 0, 0; while(nd()) // inv: a ≥ 0 ∧ b ≥ 0; { a := a + b; b++; } assert(a ≥ 0); </pre>	<pre> a, b, c := 0, 0, 0; while(nd()) // inv: b = c; { a++; b++; c++; } assert(a ≥ 100 ⇒ b = c); </pre>
---	---	---

(a) myopic generalization (b) excessive generalization (c) stuck in a rut

Figure 2.1: Verification tasks to illustrate sources of divergence for SPACER. The call *nd()* non-deterministically returns a Boolean value.

chapter.

This chapter was published as a full paper in the 32nd International Conference on Computer Aided Verification (CAV 2020) conference [72] and as an extended paper in the Formal Methods in System Design (FMSD) journal 2023 [100].

2.1 Pitfalls of Local reasoning

We illustrate the pitfalls of local reasoning using three examples shown in Fig. 2.1. All three examples are small, simple, and have simple inductive invariants. All three are challenging for SPACER. Where these examples are based on SPACER-specific design choices, each exhibits a fundamental deficiency that stems from local reasoning. We believe they can be adapted for any other IC3-style verification algorithm. The examples assume basic familiarity with the IC3 paradigm.

Myopic generalization. SPACER diverges on the example in Fig. 2.1a by iteratively learning lemmas of the form $(a - c \leq k) \Rightarrow (b - d \leq k)$ for different values of k , where a, b, c, d are the program variables. These lemmas establish that there are no counterexamples of longer and longer lengths. However, the process never converges to the desired lemma $(a - c) \leq (b - d)$, which excludes counterexamples of any length. The lemmas are discovered using interpolation, based on proofs found by the SMT-solver. A close examination of the corresponding proofs shows that the relationship between $(a - c)$ and $(b - d)$ does not appear in the proofs, making it impossible to find the desired lemma by tweaking local interpolation reasoning. On the other hand, looking at the global proof (i.e., the set of lemmas discovered to refute

a bounded counterexample), it is almost obvious that $(a - c) \leq (b - d)$ is an interesting generalization to try. Amusingly, a small, syntactic, but semantic preserving change of swapping line 2 for line 3 in Fig. 2.1a changes the SMT-solver proofs, affects local interpolation, and makes the instance trivial for SPACER.

Excessive (predecessor) generalization. SPACER diverges on the example in Fig. 2.1b by computing an infinite sequence of lemmas of the form $a + k_1 \times b \geq k_2$, where a and b are program variables, and k_1 and k_2 are integers. The root cause is excessive generalization in predecessor computation. The *Bad* states are $a < 0$, and their predecessors are states such as $(a \approx 1 \wedge b \approx -10)$, $(a \approx 2 \wedge b \approx -10)$, etc., or, more generally, regions $(a + b < 0)$, $(a + 2b < -1)$, etc. SPACER always attempts to compute the most general predecessor states. This is the best local strategy, but blocking these regions by learning their negation leads to the aforementioned lemmas. According to the global proof these lemmas do not converge to a linear invariant. An alternative strategy that under-approximates the problematic regions by (numerically) simpler regions and, as a result, learns simpler lemmas is desired (and is effective on this example). For example, region $a + 3b \leq -4$ can be under-approximated by $a \leq 32 \wedge b \leq -12$, eventually leading to a lemma $b \geq 0$, that is a part of the final invariant: $(a \geq 0 \wedge b \geq 0)$.

Stuck in a rut. Finally, SPACER converges on the example in Fig. 2.1c, but only after unrolling the system for 100 iterations. During the first 100 iterations, SPACER learns that program states with $(a \geq 100 \wedge b \not\approx c)$ are not reachable because a is bounded by 1 in the first iteration, by 2 in the second, and so on. In each iteration, the global proof is updated by replacing a lemma of the form $a < k$ by lemma of the form $a < (k + 1)$ for different values of k . Again, the strategy is good locally – total number of lemmas does not grow and the bounded proof is improved. Yet, globally, it is clear that no progress is made since the same set of bad states are blocked again and again in slightly different ways. An alternative strategy is to abstract the literal $a \geq 100$ from the formula that represents the bad states, and, instead, conjecture that no states in $b \not\approx c$ are reachable.

Our approach: global guidance. As shown in the examples above, in all the cases that SPACER diverges, the missteps are not obvious locally, but are clear when the overall proof is considered. We propose three new rules, **Subsume**, **Concretize**, and, **Conjecture**, that provide global guidance, by considering existing lemmas, to mitigate the problems illustrated above. **Subsume** introduces a lemma that generalizes existing ones, **Concretize** under-approximates partially-blocked predecessors to focus on repeatedly unblocked regions, and **Conjecture** over-approximates a predecessor

Algorithm 4: Global guidance rules for SPACER.

Subsume $\llbracket \mathcal{L} \subseteq \mathcal{O}_i, k \geq i, \mathcal{F}(\mathcal{O}_k) \Rightarrow \psi', \forall \ell \in \mathcal{L}. \psi \Rightarrow \ell \rrbracket$
 $\mathcal{O}_j := (\mathcal{O}_j \wedge \psi)$ **for all** $j \leq k + 1$
Concretize $\llbracket \mathcal{L} \subseteq \mathcal{O}_i, \langle \varphi, j \rangle \in Q, \forall \ell \in \mathcal{L}. \text{ISAT}(\varphi \wedge \neg \ell),$
 $\text{ISAT}(\varphi \wedge \bigwedge \mathcal{L}), \gamma \Rightarrow \varphi, \text{ISAT}(\gamma \wedge \bigwedge \mathcal{L}) \rrbracket$
 $Q := Q \cup \langle \gamma, k + 1 \rangle$ **where** $k = \max\{j \mid \mathcal{O}_j \Rightarrow \neg \gamma\}$
Conjecture $\llbracket \mathcal{L} \subseteq \mathcal{O}_i, \langle \varphi, j \rangle \in Q, \varphi \equiv \alpha \wedge \beta,$
 $\forall \ell \in \mathcal{L}. \ell \Rightarrow \neg \beta \wedge \text{ISAT}(\ell \wedge \alpha), R \Rightarrow \neg \alpha \rrbracket$
 $Q := Q \cup \langle \alpha, k + 1 \rangle$ **where** $k = \max\{j \mid \mathcal{O}_j \Rightarrow \neg \alpha\}$

by abstracting away regions that are repeatedly blocked. The rules are generic, and apply to arbitrary SMT theories. Furthermore, we propose an efficient instantiation of the rules for the theory Linear Integer Arithmetic.

2.2 Global guidance rules

As illustrated by the examples in Fig. 2.1, while SPACER is generally effective, its local reasoning is easily confused. The effectiveness is very dependent on the local computation of predecessors using model-based projection, and lemmas using interpolation. In this section, we extend SPACER with three additional *global* reasoning rules. The rules are inspired by the deficiencies illustrated by the motivating examples in Fig. 2.1. In this section, we present the rules abstractly, independent of any underlying theory, focusing on pre- and post-conditions. The new global rules are summarized in Alg. 4. We use the same guarded command notation as in description of SPACER in Alg. 1. Note that the rules supplement, and not replace, the ones in Alg. 1.

Subsume is the most natural rule to explain. It says that if there is a set of lemmas $\mathcal{L}$ at level i , and there exists a formula ψ such that (a) ψ is stronger than every lemma in $\mathcal{L}$, and (b) ψ over-approximates states reachable in at most k steps, where $k \geq i$, then ψ can be added to the trace to subsume $\mathcal{L}$. This rule reduces the size of the global proof – that is, the number of total not-subsumed lemmas. Note that the rule allows ψ to be at a level k that is higher than i . The choice of ψ is left open. The details are likely to be specific to the theory involved. For example, when instantiated for LIA, **Subsume** is sufficient to solve example in Fig. 2.1a. Interestingly, **Subsume** is not likely to be effective for propositional IC3. In that case, ψ is a clause and the

only way for it to be stronger than $\mathcal{L}$ is for ψ to be a syntactic sub-sequence of every lemma in $\mathcal{L}$, but such ψ is already explored by local inductive generalization (rule **Induction** in Alg. 1).

Concretize applies to a POB, unlike **Subsume**. It is motivated by example in Fig. 2.1b that highlights the problem of excessive local generalization. SPACER always computes as general predecessors as possible. This is necessary for refutational completeness since in an infinite state system there are infinitely many potential predecessors. Computing the most general predecessor ensures that SPACER finds a counterexample, if it exists. However, this also forces SPACER to discover more general, and sometimes more complex, lemmas than might be necessary for an inductive invariant. Without a global view of the overall proof, it is hard to determine when the algorithm generalizes too much. The intuition for **Concretize** is that generalization is excessive when there is a single POB $\langle \varphi, j \rangle$ that is not blocked, yet, there is a set of lemmas $\mathcal{L}$ such that every lemma $\ell \in \mathcal{L}$ partially blocks φ . That is, for any $\ell \in \mathcal{L}$, there is a sub-region φ_ℓ of POB φ that is blocked by ℓ (i.e., $\ell \Rightarrow \neg \varphi_\ell$), and there is at least one state $s \in \varphi$ that is not blocked by any existing lemma in $\mathcal{L}$ (i.e., $s \models \varphi \wedge \bigwedge \mathcal{L}$). In this case, **Concretize** computes an under-approximation γ of φ that includes some not-yet-blocked state s . The new POB is added to the lowest level at which γ is not yet blocked. **Concretize** is useful to solve the example in Fig. 2.1b.

Conjecture guides the algorithm away from being stuck in the same part of the search space. A single POB φ might be blocked by a different lemma at each level that φ appears in. This indicates that the lemmas are too strong, and cannot be propagated successfully to a higher level. The goal of the **Conjecture** rule is to identify such a case to guide the algorithm to explore alternative proofs with a better potential for generalization. This is done by abstracting away the part of the POB that has been blocked in the past. The pre-condition for **Conjecture** is the existence of a POB $\langle \varphi, j \rangle$ such that φ is split into two (not necessarily disjoint) sets of literals, α and β . Second, there must be a set of lemmas $\mathcal{L}$, at a (typically much lower) level $i < j$ such that every lemma $\ell \in \mathcal{L}$ blocks φ , and, moreover, blocks φ by blocking β . Intuitively, this implies that while there are many different lemmas (i.e., all lemmas in $\mathcal{L}$) that block φ at different levels, all of them correspond to a *local* generalization of $\neg \beta$ that could not be propagated to block φ at higher levels. In this case, **Conjecture** abstracts the POB φ into α , hoping to generate an alternative way to block φ . Of course, α is conjectured only if it is not already blocked and does not contain any known reachable states. **Conjecture** is necessary for a quick convergence on the example in Fig. 2.1c. In some respect, **Conjecture** is akin to widening in Abstract Interpretation [25] – it abstracts a set of states by dropping constraints that appear to prevent further exploration.

Of course, it is also quite different since it does not guarantee termination. While **Conjecture** is applicable to propositional IC3 as well, it is much more significant in SMT-based setting since in many FOL theories a single literal in a POB might result in infinitely many distinct lemmas.

Each of the rules can be applied by itself, but they are most effective in combination. For example, **Concretize** creates less general predecessors, that, in the worst case, lead to many simple lemmas. At the same time, **Subsume** combines lemmas together into more complex ones. The interaction of the two produces lemmas that neither one can produce in isolation. At the same time, **Conjecture** helps unstuck the algorithm from a single unproductive POB, allowing the other rules to take effect.

2.3 Global guidance for Linear Integer Arithmetic

In this section, we present a specialization of our general rules, shown in Alg. 4, to the theory of Linear Integer Arithmetic (LIA). This requires solving two problems: identifying subsets of lemmas for pre-conditions of the rules (clearly using all possible subsets is too expensive), and applying the rule once its pre-condition is met. For lemma selection, we introduce a notion of syntactic clustering based on anti-unification. For rule application, we exploit basic properties of LIA for an effective algorithm. Our presentation is focused on LIA exclusively. However, the rules extend to combinations of LIA with other theories, such as the combined theory of LIA and Arrays.

The rest of this section is structured as follows. We present our lemma selection scheme, which is common to all the rules, in Sec. 2.3.1, followed by a description of how the rules **Subsume** (in Sec. 2.3.2), **Concretize** (in Sec. 2.3.3), and **Conjecture** (in Sec. 2.3.4) are instantiated for LIA. We conclude in Sec. 2.3.5 with an algorithm that integrates all the rules together.

2.3.1 Lemma selection

A common pre-condition for all of our global rules in Alg. 4 is the existence of a subset of lemmas $\mathcal{L}$ of some frame $\mathcal{O}_i$. Attempting to apply the rules for every subset of $\mathcal{O}_i$ is infeasible. In practice, we use syntactic similarity between lemmas as a predictor that one of the global rules is applicable, and restrict $\mathcal{L}$ to subsets of syntactically similar lemmas. In the rest of this section, we formally define what we mean by *syntactic similarity*, and how syntactically similar subsets of lemmas, called *clusters*, are maintained efficiently throughout the algorithm.

Syntactic similarity. A formula π with free variables is called a *pattern*. Note that we do not require π to be in LIA. Let σ be a substitution, i.e., a mapping from variables to terms. We write $\pi\sigma$ for the result of replacing all occurrences of free variables in π with their mapping under σ . A substitution σ is called *numeric* if it maps every variable to an integer, i.e., the range of σ is $\mathbb{Z}$. We say that a formula φ *numerically matches* a pattern π iff there exists a numeric substitution σ such that $\varphi = \pi\sigma$. Note that, as usual, the equality is syntactic. For example, consider the pattern $\pi = v_0a + v_1b \leq 0$ with free variables v_0 and v_1 and uninterpreted constants a and b . The formula $\varphi_1 = 3a + 4b \leq 0$ matches π via a numeric substitution $\sigma_1 = \{v_0 \mapsto 3, v_1 \mapsto 4\}$. However, $\varphi_2 = 4b + 3a \leq 0$, while semantically equivalent to φ_1 , does not match π . Similarly $\varphi_3 = a + b \leq 0$ does not match π as well.

Matching is extended to patterns in the usual way by allowing a substitution σ to map variables to variables. We say that a pattern π_1 is more general than a pattern π_2 if π_2 matches π_1 . A pattern π is a *numeric anti-unifier* for a pair of formulas φ_1 and φ_2 if both φ_1 and φ_2 match π numerically. We write $anti(\varphi_1, \varphi_2)$ for a most general numeric anti-unifier of φ_1 and φ_2 . We say that two formulas φ_1 and φ_2 are *syntactically similar* if there exists a numeric anti-unifier between them (i.e., $anti(\varphi_1, \varphi_2)$ is defined). Anti-unification is extended to sets of formulas in the usual way.

Clusters. We use anti-unification to define *clusters* of syntactically similar formulas. Let Φ be a fixed set of formulas, and π a pattern. A *cluster*, $\mathcal{C}_\Phi(\pi)$, is a subset of Φ such that every formula $\varphi \in \mathcal{C}_\Phi(\pi)$ numerically matches π . That is, π is a numeric anti-unifier for $\mathcal{C}_\Phi(\pi)$. In the implementation, we restrict the pre-conditions of the global rules so that a subset of lemmas $\mathcal{L} \subseteq \mathcal{O}_i$ is a cluster for some pattern π , i.e., $\mathcal{L} = \mathcal{C}_{\mathcal{O}_i}(\pi)$.

Clustering lemmas. We use the following strategy to efficiently keep track of available clusters. Let ℓ_{new} be a new lemma to be added to $\mathcal{O}_i$. Assume there is at least one lemma $\ell \in \mathcal{O}_i$ that numerically anti-unifies with ℓ_{new} via some pattern π . If such an ℓ does not belong to any cluster, a new cluster $\mathcal{C}_{\mathcal{O}_i}(\pi) = \{\ell_{\text{new}}, \ell\}$ is formed, where $\pi = anti(\ell_{\text{new}}, \ell)$. Otherwise, for every lemma $\ell \in \mathcal{O}_i$ that numerically matches ℓ_{new} and every cluster $\mathcal{C}_{\mathcal{O}_i}(\hat{\pi})$ containing ℓ , ℓ_{new} is added to $\mathcal{C}_{\mathcal{O}_i}(\hat{\pi})$ if ℓ_{new} matches $\hat{\pi}$, or a new cluster is formed using ℓ , ℓ_{new} , and any other lemmas in $\mathcal{C}_{\mathcal{O}_i}(\hat{\pi})$ that anti-unify with them. Note that a new lemma ℓ_{new} might belong to multiple clusters.

For example, suppose $\ell_{\text{new}} = (a \leq 6 \vee b \leq 6)$, and there is already a cluster $\mathcal{C}_{\mathcal{O}_i}(a \leq v_0 \vee b \leq 5) = \{(a \leq 5 \vee b \leq 5), (a \leq 8 \vee b \leq 5)\}$. Since ℓ_{new} anti-unifies with each of the lemmas in the cluster, but does not match the pattern $a \leq v_0 \vee b \leq 5$,

a new cluster that includes all of them is formed w.r.t. a more general pattern: $\mathcal{C}_{O_i}(a \leq v_0 \vee b \leq v_1) = \{(a \leq 6 \vee b \leq 6), (a \leq 5 \vee b \leq 5), (a \leq 8 \vee b \leq 5)\}$.

In the presentation above, we assumed that anti-unification (and matching) is purely syntactic, based on syntactic substitution (as defined by Plotkin [90]). This allows us to compute anti-unification of two formulas in linear time. However, this is problematic in practice since it significantly limits the applicability of the global rules. Recall, for example, that $a + b \leq 0$ and $2a + 2b \leq 0$ do not anti-unify numerically according to our definitions, and, therefore, do not cluster together. In practice, we augment syntactic anti-unification with simple rewrite rules that are applied greedily. For example, we normalize all LIA terms, take care of implicit multiplication by 1, and of associativity and of commutativity addition. We do not claim any completeness results for anti-unification modulo associativity or commutativity. In the future, it is interesting to explore how advanced anti-unification algorithms, such as [18], [104], can be adapted for our purpose.

2.3.2 Subsume rule for LIA

Recall that the **Subsume** rule (Alg. 4) takes a cluster of lemmas $\mathcal{L} = \mathcal{C}_{O_i}(\pi)$ and computes a new lemma ψ that subsumes all the lemmas in $\mathcal{L}$, that is $\psi \Rightarrow \bigwedge \mathcal{L}$. We find it convenient to dualize the problem. Let $\mathcal{S} = \{\neg \ell \mid \ell \in \mathcal{L}\}$ be the dual of $\mathcal{L}$, clearly $\psi \Rightarrow \bigwedge \mathcal{L}$ iff $(\bigvee \mathcal{S}) \Rightarrow \neg \psi$. Note that $\mathcal{L}$ is a set of clauses, $\mathcal{S}$ is a set of cubes, ψ is a clause, and $\neg \psi$ is a cube. In the case of $\text{LIA}^{-\text{div}}$ (defined in Sec. 1.2.2), this means that $\bigvee \mathcal{S}$ represents a union of convex sets, and $\neg \psi$ represents a convex set that the **Subsume** rule must find. The strongest such $\neg \psi$ in $\text{LIA}^{-\text{div}}$ exists, and is the convex closure of $\mathcal{S}$. Thus, applying **Subsume** in the context of $\text{LIA}^{-\text{div}}$ is reduced to computing a convex closure of a set of (negated) lemmas in a cluster. Full LIA extends $\text{LIA}^{-\text{div}}$ with divisibility constraints. Therefore, **Subsume** obtains a stronger $\neg \psi$ by adding such constraints.

Example 5 For example, consider the following cluster:

$$\begin{aligned}\mathcal{L} &= \{(x > 2 \vee x < 2 \vee y > 3), (x > 4 \vee x < 4 \vee y > 5), (x > 8 \vee x < 8 \vee y > 9)\} \\ \mathcal{S} &= \{(x \leq 2 \wedge x \geq 2 \wedge y \leq 3), (x \geq 4 \wedge x \leq 4 \wedge y \leq 5), (x \geq 8 \wedge x \leq 8 \wedge y \leq 9)\}\end{aligned}$$

The convex closure of $\mathcal{S}$ in $\text{LIA}^{-\text{div}}$ is $2 \leq x \leq 8 \wedge y \leq x + 1$. However, a stronger over-approximation exists in LIA: $2 \leq x \leq 8 \wedge y \leq x + 1 \wedge (2 \mid x)$. ■

In the sequel, we describe **SUBSUMECUBE** (Alg. 5) which computes a cube φ that over-approximates $(\bigvee \mathcal{S})$. **Subsume** is then implemented by removing from $\mathcal{L}$

Algorithm 5: An implementation of the Subsume rule for the dual of a cluster
 $\mathcal{S} = \{A \cdot \bar{x} \leq \bar{n}_i \mid 1 \leq i \leq q\}$.

```

function SUBSUMECUBE:
  In:  $\mathcal{S} = \{(A \cdot \bar{x} \leq \bar{n}_i) \mid 1 \leq i \leq q\}$ ,
  Out: An over-approximation of  $(\bigvee \mathcal{S})$ .
  /*  $\bar{v}$  are integer variables such that:
     $(\bigvee \mathcal{S}) \iff \exists \bar{v}. (A \cdot \bar{x} \leq \bar{v}) \wedge (\bigvee \bar{v} \approx \bar{n}_i)$  */
1   $N := [\bar{n}_1; \dots; \bar{n}_q]^T$ 
  /* Compute the set of linear dependencies implied by  $N$  */
2   $B := \text{kernel}([N; \bar{1}])$ 
3   $L := \bigwedge_{\bar{y} \in B} (v_1 \dots v_p \ 1) \cdot \bar{y} \approx 0$ 
4  if  $|\bar{v}^{L_\downarrow}| = 1$  then
    // Convex closure over a single constant  $v_i \in \bar{v}^{L_\downarrow}$ 
5     $C := \min(N_{*i}) \leq v_i \leq \max(N_{*i})$ 
6  else
    // Syntactic convex closure
7     $C := (\bar{\alpha}^T \cdot N^{L_\downarrow} \approx (\bar{v}^{L_\downarrow})^T) \wedge (\sum \bar{\alpha} \approx 1) \wedge (\bar{\alpha} \geq 0)$ 
    /* Compute divisibility constraints */
8   $D := \top$ 
9  for  $v_j \in \bar{v}^{L_\downarrow}$  do
10    if  $\exists d, r. d \not\approx 1 \wedge (\forall n \in N_{*j}^{L_\downarrow}. (n \bmod d \approx r))$  then
11       $D := D \wedge d \mid (v_j - r)$ 
12   $\psi := (A \cdot \bar{x} \leq \bar{v}) \wedge L \wedge C \wedge D$ 
    /* Under-approximate quantifier elimination */
13  find  $M_0$  s.t.  $M_0 \models \psi$  and, if possible,  $M_0 \not\models (\bigvee \mathcal{S})$ 
14   $\varphi := \text{MBP}((\bar{\alpha} \bar{v}), \psi, M_0)$ 
    /* Over-approximate quantifier elimination */
15  while  $\text{ISSAT}(\neg \varphi \wedge \psi)$  do
16    find  $M_1$  s.t.  $M_1 \models (\neg \varphi \wedge \psi)$ 
17     $\varphi := \bigwedge \{\ell \in \varphi \mid \neg(M_1 \models \neg \ell)\}$ 
18  return  $\varphi$ 

```

lemmas that are already subsumed by existing lemmas in $\mathcal{L}$, dualizing the result into $\mathcal{S}$, invoking SUBSUMECUBE on $\mathcal{S}$ and returning $\neg \varphi$ as a lemma that subsumes $\mathcal{L}$.

Recall that Subsume is tried only in the case $\mathcal{L} = \mathcal{C}_{\mathcal{O}_i}(\pi)$. We further require that the negated pattern, $\neg \pi$, is of the form $A \cdot \bar{x} \leq \bar{v}$, where A is a coefficients matrix, $\bar{x}$ is a vector of constants and $\bar{v} = (v_1 \dots v_p)^T$ is a vector of p free variables. Under this

assumption, $\mathcal{S}$ (the dual of $\mathcal{L}$) is of the form $\{(A \cdot \bar{x} \leq \bar{n}_i) \mid 1 \leq i \leq q\}$, where $q = |\mathcal{S}|$, and for each $1 \leq i \leq q$, $\bar{n}_i$ is a numeric substitution to $\bar{v}$ from which one of the negated lemmas in $\mathcal{S}$ is obtained. That is, $|\bar{n}_i| = |\bar{v}|$. In Ex. 5, $\neg\pi = x \leq v_1 \wedge -x \leq v_2 \wedge y \leq v_3$ and

$$A = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \end{bmatrix} \quad \bar{x} = \begin{bmatrix} x \\ y \end{bmatrix} \quad \bar{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \quad \bar{n}_1 = \begin{bmatrix} 2 \\ -2 \\ 3 \end{bmatrix} \quad \bar{n}_2 = \begin{bmatrix} 4 \\ -4 \\ 5 \end{bmatrix} \quad \bar{n}_3 = \begin{bmatrix} 8 \\ -8 \\ 9 \end{bmatrix}$$

Each cube $(A \cdot \bar{x} \leq \bar{n}_i) \in \mathcal{S}$ is equivalent to $\exists \bar{v}. A \cdot \bar{x} \leq \bar{v} \wedge (\bar{v} \approx \bar{n}_i)$. Finally, $(\bigvee \mathcal{S}) \equiv \exists \bar{v}. (A \cdot \bar{x} \leq \bar{v}) \wedge (\bigvee (\bar{v} \approx \bar{n}_i))$. Thus, computing the over-approximation of $\mathcal{S}$ is reduced to (a) computing the convex hull H of a set of points $\{\bar{n}_i \mid 1 \leq i \leq q\}$, (b) computing divisibility constraints D that are satisfied by all the points, (c) substituting $H \wedge D$ for the disjunction in the equation above, and (c) eliminating variables $\bar{v}$. Both the computation of $H \wedge D$ and the elimination of $\bar{v}$ may be prohibitively expensive. We, therefore, over-approximate them. Our approach for doing so is presented in Alg. 5, and explained in detail below.

Computing the convex hull of $\{\bar{n}_i \mid 1 \leq i \leq q\}$. lines 2 to 7 compute the convex hull of $\{\bar{n}_i \mid 1 \leq i \leq q\}$ as a formula over $\bar{v}$, where variable v_j , for $1 \leq j \leq p$, represents the j^{th} coordinates in the vectors (points) $\bar{n}_i$. Some of the coordinates, v_j , in these vectors may be linearly dependent upon others. To simplify the problem, we first identify such dependencies and compute a set of linear equalities that expresses them (L in line 3). To do so, we consider a matrix $N_{q \times p}$, where the i^{th} row consists of $\bar{n}_i^T$. The j^{th} column in N , denoted N_{*j} , corresponds to the j^{th} coordinate, v_j . The rank of N is the number of linearly independent columns (and rows). The other columns (coordinates) can be expressed by linear combinations of the linearly independent ones. To compute these linear combinations we use the kernel of $[N; \bar{1}]$ (N appended with a column vector of 1's), which is the set of all vectors $\bar{y}$ such that $[N; \bar{1}] \cdot \bar{y} \approx \bar{0}$, where $\bar{0}$ is the zero vector. Let $B = \text{kernel}([N; \bar{1}])$ be a basis for the kernel of $[N; \bar{1}]$. Then $|B| = p - \text{rank}(N)$, and for each vector $\bar{y} \in B$, the linear equality $[v_1 \cdots v_p \ 1] \cdot \bar{y} \approx 0$ holds in all the rows of N (i.e., all the given vectors satisfy it). We accumulate these equalities, which capture the linear dependencies between the coordinates, in L . Further, the equalities are used to compute $\text{rank}(N)$ coordinates (columns in N) that are linearly independent and, modulo L , uniquely determine the remaining coordinates. We denote by $\bar{v}^{L\downarrow}$ the subset of $\bar{v}$ that consists of the linearly independent coordinates. We further denote by $\bar{n}_i^{L\downarrow}$ the projection of $\bar{n}_i$ to these coordinates and by $N^{L\downarrow}$ the projection of N to the corresponding columns. We have that $(\bigvee (\bar{v} \approx \bar{n}_i)) \equiv L \wedge (\bigvee (\bar{v}^{L\downarrow} \approx \bar{n}_i^{L\downarrow}))$.

In Ex. 5, the numeral matrix is $N = \begin{bmatrix} 2 & -2 & 3 \\ 4 & -4 & 5 \\ 8 & -8 & 9 \end{bmatrix}$, for which $\text{kernel}([N; \bar{1}]) = \{(1 \ 1 \ 0 \ 0)^T, (1 \ 0 \ -1 \ 1)^T\}$. Therefore, L is the conjunction of equalities $v_1 + v_2 \approx 0 \wedge v_1 - v_3 + 1 \approx 0$, or, equivalently $v_3 \approx v_1 + 1 \wedge v_2 \approx -v_1$, $\bar{v}^{L\downarrow} = \begin{pmatrix} v_1 \end{pmatrix}^T$, and

$$\bar{n}_1^{L\downarrow} = \begin{bmatrix} 2 \end{bmatrix} \quad \bar{n}_2^{L\downarrow} = \begin{bmatrix} 4 \end{bmatrix} \quad \bar{n}_3^{L\downarrow} = \begin{bmatrix} 8 \end{bmatrix} \quad N^{L\downarrow} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix}$$

Next, we compute the convex closure of $\bigvee(\bar{v}^{L\downarrow} \approx \bar{n}_i^{L\downarrow})$, and conjoin it with L to obtain H , the convex closure of $(\bigvee(\bar{v} \approx \bar{n}_i))$.

If the dimension of $\bar{v}^{L\downarrow}$ is one, as is the case in the example above, convex closure, C , of $\bigvee(\bar{v}^{L\downarrow} \approx \bar{n}_i^{L\downarrow})$ is obtained by bounding the sole element of $\bar{v}^{L\downarrow}$ based on its values in $N^{L\downarrow}$ (line 5). In Ex. 5, we obtain $C = 2 \leq v_1 \leq 8$.

If the dimension of $\bar{v}^{L\downarrow}$ is greater than one, just computing the bounds of one of the constants is not sufficient. Instead, we use the concept of syntactic convex closure from [10] to compute the convex closure of $\bigvee(\bar{v}^{L\downarrow} \approx \bar{n}_i^{L\downarrow})$ as $\exists \bar{\alpha}. C$ where $\bar{\alpha}$ is a vector that consists of q fresh *rational* variables and C is defined as follows (line 7): $C = \bar{\alpha} \geq 0 \wedge \Sigma \bar{\alpha} \approx 1 \wedge \bar{\alpha}^T \cdot N^{L\downarrow} \approx (\bar{v}^{L\downarrow})^T$. C states that $(\bar{v}^{L\downarrow})^T$ is a convex combination of the rows of $N^{L\downarrow}$, or, in other words, $\bar{v}^{L\downarrow}$ is a convex combination of $\{\bar{n}_i^{L\downarrow} \mid 1 \leq i \leq q\}$.

To illustrate the syntactic convex closure, consider a second example with a set of cubes: $\mathcal{S} = \{(x \leq 0 \wedge y \leq 6), (x \leq 6 \wedge y \leq 0), (x \leq 5 \wedge y \leq 5)\}$. The coefficient matrix A , and the numeral matrix N are then: $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $N = \begin{bmatrix} 0 & 6 \\ 6 & 0 \\ 5 & 5 \end{bmatrix}$. Here, $\text{kernel}([N; \bar{1}])$ is empty – all the columns are linearly independent, hence, $L = \text{true}$ and $\bar{v}^{L\downarrow} = \bar{v}$. Therefore, syntactic convex closure is applied to the full matrix N , resulting in

$$C = (\alpha_1 \geq 0) \wedge (\alpha_2 \geq 0) \wedge (\alpha_3 \geq 0) \wedge (\alpha_1 + \alpha_2 + \alpha_3 \approx 1) \wedge \\ (6\alpha_2 + 5\alpha_3 \approx v_1) \wedge (6\alpha_1 + 5\alpha_3 \approx v_2)$$

The convex closure of $\bigvee(\bar{v} \approx \bar{n}_i)$ is then $L \wedge \exists \bar{\alpha}. C$, which is $\exists \bar{\alpha}. C$ here.

Divisibility constraints. Inductive invariants for verification problems often require divisibility constraints. We, therefore, use such constraints, denoted D , to obtain a stronger over-approximation of $\bigvee(\bar{v} \approx \bar{n}_i)$ than the convex closure. To add a divisibility constraint for $v_j \in \bar{v}^{L\downarrow}$, we consider the column $N_{*j}^{L\downarrow}$ that corresponds to v_j in $N^{L\downarrow}$. We find the largest positive integer d such that each integer in $N_{*j}^{L\downarrow}$ leaves

the same remainder when divided by d ; namely, there exists $0 \leq r < d$ such that $n \bmod d \approx r$ for every $n \in N_{*j}^{L_\downarrow}$. This means that $d \mid (v_j - r)$ is satisfied by all the points $\bar{n}_i$. Note that such r always exists for $d \approx 1$. To avoid this trivial case, we add the constraint $d \mid (v_j - r)$ only if $d \not\approx 1$ (line 11). We repeat this process for each $v_j \in \bar{v}^{L_\downarrow}$.

In Ex. 5, all the elements in the (only) column of the matrix $N^{L_\downarrow}$, which corresponds to v_1 , are divisible by 2, and no larger d has a corresponding r . Thus, line 11 of Alg. 5 adds the divisibility condition $(2 \mid v_1)$ to D .

Eliminating existentially quantified variables using MBP. By combining the linear equalities exhibited by N , the convex closure of $N^{L_\downarrow}$ and the divisibility constraints on $\bar{v}$, we obtain $\exists \bar{\alpha}. L \wedge C \wedge D$ as an over-approximation of $\bigvee (\bar{v} \approx \bar{n}_i)$. Accordingly, $\exists \bar{v}. \exists \bar{\alpha}. \psi$, where $\psi = (A \cdot \bar{x} \leq \bar{v}) \wedge L \wedge C \wedge D$, is an over-approximation of $(\bigvee \mathcal{S}) \equiv \exists \bar{v}. (A \cdot \bar{x} \leq \bar{v}) \wedge (\bigvee (\bar{v} \approx \bar{n}_i))$ (line 12). In order to get a LIA cube that overapproximates $\bigvee \mathcal{S}$, it remains to eliminate the existential quantifiers. Since quantifier elimination is expensive, and does not necessarily generate convex formulas (cubes), we approximate it using MBP. Namely, we obtain a cube φ that under-approximates $\exists \bar{v}. \exists \bar{\alpha}. \psi$ by applying MBP on ψ and a model $M_0 \models \psi$. We then use an SMT solver to drop literals from φ until it over-approximates $\exists \bar{v}. \exists \bar{\alpha}. \psi$, and hence also $\bigvee \mathcal{S}$ (lines 15 to 18). The result is returned by `Subsume` as an over-approximation of $\bigvee \mathcal{S}$.

Models M_0 that satisfy ψ and do not satisfy any of the cubes in $\mathcal{S}$ are preferred when computing MBP (line 13) as they ensure that the result of MBP is not subsumed by any of the cubes in $\mathcal{S}$.

Note that the $\bar{\alpha}$ are rational variables and $\bar{v}$ are integer variables, which means we require MBP to support a mixture of integer and rational variables. To achieve this, we first relax all constants to be rationals and apply MBP over LRA to eliminate $\bar{\alpha}$. We then adjust the resulting formula back to integer arithmetic by multiplying each atom by the least common multiple of the denominators of the coefficients in it. Finally, we apply MBP over the integers to eliminate $\bar{v}$.

Considering Ex. 5 again, we get that $\psi = (x \leq v_1) \wedge (-x \leq v_2) \wedge (y \leq v_3) \wedge (v_3 \approx 1 + v_1) \wedge (v_2 \approx -v_1) \wedge (2 \leq v_1 \leq 8) \wedge (2 \mid v_1)$ (the first three conjuncts correspond to $(A \cdot (x \ y)^T \leq (v_1 \ v_2 \ v_3)^T)$). Note that in this case we do not have rational variables $\bar{\alpha}$ since $|\bar{v}^{L_\downarrow}| = 1$. Depending on the model, the result of MBP can be one of

$$\begin{aligned} y \leq x + 1 \wedge 2 \leq x \leq 8 \wedge (2 \mid y - 1) \wedge (2 \mid x) & \quad x \geq 2 \wedge x \leq 2 \wedge y \leq 3 \\ y \leq x + 1 \wedge 2 \leq x \leq 8 \wedge (2 \mid x) & \quad x \geq 8 \wedge x \leq 8 \wedge y \leq 9 \\ y \geq x + 1 \wedge y \leq x + 1 \wedge 3 \leq y \leq 9 \wedge (2 \mid y - 1) & \end{aligned}$$

However, we prefer a model that does not satisfy any cube in $\mathcal{S} = \{(x \geq 2 \wedge x \leq 2 \wedge y \leq 3), (x \leq 4 \wedge x \geq 4 \wedge y \leq 5), (x \leq 8 \wedge x \geq 8 \wedge y \leq 9)\}$, rules off the two possibilities on the right. None of these cubes cover ψ , hence generalization is used.

If the first cube is obtained by MBP, it is generalized into $y \leq x + 1 \wedge x \geq 2 \wedge x \leq 8 \wedge (2 \mid x)$; the second cube is already an over-approximation; the third cube is generalized into $y \leq x + 1 \wedge y \leq 9$. Indeed, each of these cubes over-approximates $\bigvee \mathcal{S}$.

2.3.3 Concretize rule for LIA

The **Concretize** rule (Alg. 4) takes a cluster of lemmas $\mathcal{L} = \mathcal{C}_{\mathcal{O}_i}(\pi)$ and a POB $\langle \varphi, j \rangle$ such that each lemma in $\mathcal{L}$ partially blocks φ , and creates a new POB γ that is still not blocked by $\mathcal{L}$, but γ is more concrete, i.e., $\gamma \Rightarrow \varphi$. In our implementation, this rule is applied when φ is in LIA^{div} . We further require that the pattern, π , of $\mathcal{L}$ is non-linear, i.e., some of the constants appear in π with free variables as their coefficients. We denote these constants by U . An example is the pattern $\pi = v_0x + v_1y + z \leq 0$, where $U = \{x, y\}$. Having such a cluster is an indication that attempting to block φ in full with a single lemma may require to track non-linear correlations between the constants, which is impossible to do in LIA. In such cases, we identify the coupling of the constants in U in POBs (and hence in lemmas) as the potential source of non-linearity. Hence, we concretize (strengthen) φ into a POB γ where the constants in U are no longer coupled to any other constant.

Coupling. Formally, constants u and v are *coupled* in a cube c , denoted $u \bowtie_c v$, if there exists a literal *lit* in c such that both u and v appear in *lit* (i.e., their coefficients in *lit* are non-zero). For example, x and y are coupled in $x + y \leq 0 \wedge z \leq 0$ whereas neither of them are coupled with z . A constant u is said to be *isolated* in a cube c , denoted $\text{ISO}(u, c)$, if it appears in c but it is not coupled with any other constant in c . In the above cube, z is isolated.

Concretization by decoupling. Given a POB φ (a cube) and a cluster $\mathcal{L}$, Alg. 6 presents our approach for concretizing φ by decoupling the constants in U — those that have variables as coefficients in the pattern of $\mathcal{L}$ (line 1). Concretization is guided by a model $M \models \varphi \wedge \bigwedge \mathcal{L}$, representing a part of φ that is not yet blocked by the lemmas in $\mathcal{L}$ (line 2). Given such M , we concretize φ into a *model-preserving* under-approximation that isolates all the constants in U and preserves all other couplings. That is, we find a cube γ , such that

$$\gamma \Rightarrow \varphi \quad M \models \gamma \quad \forall u \in U. \text{ISO}(u, \gamma) \quad \forall u, v \notin U. (u \bowtie_{\varphi} v) \Rightarrow (u \bowtie_{\gamma} v) \quad (2.1)$$

Algorithm 6: An implementation of the Concretize rule in LIA.

```

function CONCRETIZE:
  In: A POB  $\langle \varphi, j \rangle$  in  $\text{LIA}^{-\text{div}}$ , a cluster of  $\text{LIA}^{-\text{div}}$  lemmas  $\mathcal{L} = \mathcal{C}_{\mathcal{O}_i}(\pi)$  s.t.  $\pi$ 
    is non-linear,  $\text{ISAT}(\varphi \wedge \bigwedge \mathcal{L})$ 
  Out: A cube  $\gamma$  such that  $\gamma \Rightarrow \varphi$  and  $\forall \ell \in \mathcal{L}. \text{ISAT}(\gamma \wedge \ell)$ 
  1  $U := \{x \mid \text{COEFF}(x, \pi) \in \text{VARS}(\pi)\}$ 
  2 find  $M$  s.t.  $M \models \varphi \wedge \bigwedge \mathcal{L}$ 
  3  $\gamma := \top$ 
  4 foreach  $lit \in \varphi$  do
  5   if  $\mathcal{C}lit \cap U \neq \emptyset$  then  $\gamma := \gamma \wedge \text{CONCRETIZE\_LIT}(lit, M, U)$ 
  6   else  $\gamma := \gamma \wedge lit$ 
  7  $\gamma := \text{RM\_SUBSUME}(\gamma)$ 
  8 return  $\gamma$ 

  function CONCRETIZE_LIT:
  In: A literal  $lit = \sum_i n_i x_i \leq b_j$  in  $\text{LIA}^{-\text{div}}$ , model  $M \models lit$ , and a set of
    constants  $U$ 
  Out: A cube  $\gamma^{lit}$  that concretizes  $lit$ 
  /* Construct a single literal using all the constants in  $\mathcal{C}lit \setminus U$  */
  9  $\gamma^{lit} := \emptyset$ 
  10  $s := 0$ 
  11 foreach  $x_i \in \mathcal{C}lit \setminus U$  do
  12    $s := s + n_i x_i$ 
  13  $\gamma^{lit} := (s \leq M[s])$ 
  /* Generate one dimensional literals for each constant in  $U$  */
  14 foreach  $x_i \in \mathcal{C}lit \cap U$  do
  15    $\gamma^{lit} := \gamma^{lit} \wedge (n_i x_i \leq M[n_i x_i])$ 
  16 return  $\gamma^{lit}$ 

```

Note that γ is not blocked by $\mathcal{L}$ since M satisfies both $\bigwedge \mathcal{L}$ and γ . For example, if $\varphi = (x + y \leq 0) \wedge (x - y \leq 0) \wedge (x + z \geq 0)$ and $M = [x = 0, y = 0, z = 1]$, then $\gamma = 0 \leq y \leq 0 \wedge x \leq 0 \wedge x + z \geq 1$ is a model preserving under-approximation that isolates $U = \{y\}$.

Alg. 6 computes such a cube γ by a point-wise concretization of the literals of φ followed by the removal of subsumed literals. Literals that do not contain constants from U remain unchanged. A literal of the form $lit = t \leq b$, where $t = \sum_i n_i x_i$ (recall that every literal in $\text{LIA}^{-\text{div}}$ can be normalized to this form), that includes constants from U is concretized into a *cube* by (1) isolating each of the summands

$n_i x_i$ in t that include U from the rest, and (2) for each of the resulting sub-expressions creating a literal that uses its value in M as a bound. Formally, t is decomposed to $s + \sum_{x_i \in U} n_i x_i$, where $s = \sum_{x_i \notin U} n_i x_i$. The concretization of lit is the cube $\gamma^{lit} = s \leq M[s] \wedge \bigwedge_{x_i \in U} n_i x_i \leq M[n_i x_i]$, where $M[t']$ denotes the interpretation of t' in M . Note that $\gamma^{lit} \Rightarrow lit$ since the bounds are stronger than the original bound on t : $M[s] + \sum_{x_i \in U} M[n_i x_i] = M[t] \leq b$. This ensures that γ , obtained by the conjunction of literal concretizations, implies φ . It trivially satisfies the other conditions of Eq. (2.1).

For example, the concretization of the literal $(x + y \leq 0)$ with respect to $U = \{y\}$ and $M = [x = 0, y = 0, z = 1]$ is the cube $x \leq 0 \wedge y \leq 0$. Applying concretization in a similar manner to all the literals of the cube $\varphi = (x + y \leq 0) \wedge (x - y \leq 0) \wedge (x + z \geq 0)$ from the previous example, we obtain the concretization $x \leq 0 \wedge 0 \leq y \leq 0 \wedge x + z \geq 0$. Note that the last literal is not concretized as it does not include y .

2.3.4 Conjecture rule for LIA

The **Conjecture** rule (see Alg. 4) takes a set of lemmas $\mathcal{L}$ and a POB $\varphi \equiv \alpha \wedge \beta$ such that all lemmas in $\mathcal{L}$ block β , but none of them blocks α , where α does not include any known reachable states. It returns α as a new POB.

For LIA, **Conjecture** is applied when the following conditions are met: (1) the POB φ is of the form $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$, where $\varphi_3 = (\bar{n}^T \cdot \bar{x} \leq b)$, and φ_1 and φ_2 are any cubes. The sub-cube $\varphi_1 \wedge \varphi_2$ acts as α , while the sub-cube $\varphi_2 \wedge \varphi_3$ acts as β . (2) The cluster $\mathcal{L}$ consists of $\{bg \vee (\bar{n}^T \cdot \bar{x} \geq b_i) \mid 1 \leq i \leq q\}$, where $b_i > b$ and $bg \Rightarrow \neg \varphi_2$. This means that each of the lemmas in $\mathcal{L}$ blocks $\beta = \varphi_2 \wedge \varphi_3$, and they may be ordered as a sequence of increasingly stronger lemmas, indicating that they were created by trying to block the POB at different levels, leading to too strong lemmas that failed to propagate to higher levels. (3) The formula $(bg \vee (\bar{n}^T \cdot \bar{x} \geq b_i)) \wedge \varphi_1 \wedge \varphi_2$ is satisfiable, that is, none of the lemmas in $\mathcal{L}$ block $\alpha = \varphi_1 \wedge \varphi_2$, and (4) $R \Rightarrow \neg(\varphi_1 \wedge \varphi_2)$, that is, no state in $\varphi_1 \wedge \varphi_2$ is known to be reachable. If all four conditions are met, we conjecture $\alpha = \varphi_1 \wedge \varphi_2$. This is implemented by **CONJECTURE**, that returns α (or $\perp$ when the pre-conditions are not met).

For example, consider the POB $\varphi = x \geq 10 \wedge (x + y \geq 10) \wedge y \leq 10$ and a cluster of lemmas $\mathcal{L} = \{(x + y \leq 0 \vee y \geq 101), (x + y \leq 0 \vee y \geq 102)\}$. In this case, $\varphi_1 = x \geq 10$, $\varphi_2 = (x + y \geq 10)$, $\varphi_3 = y \leq 10$, and $bg = x + y \leq 0$. Each of the lemmas in $\mathcal{L}$ block $\varphi_2 \wedge \varphi_3$ but none of them block $\varphi_1 \wedge \varphi_2$. Therefore, we conjecture $\varphi_1 \wedge \varphi_2$: $x \geq 10 \wedge (x + y \geq 10)$.

Algorithm 7: GSPACER for LIA.

<pre> function GSpacer: In: $\langle \text{Init}, \text{Tr}, \text{Bad} \rangle$ Out: An Inductive invariant or UNSAFE /* Initialize state of the solver */ 1 $Q := \emptyset; N := 0; R := \text{Init};$ 2 $\mathcal{O}_0 := \text{Init}; \mathcal{O}_i := \top, \forall i > 0$ 3 Enqueue($Q, \langle \text{Bad}, 0 \rangle$) 4 while $\top$ do 5 $\langle \varphi, i \rangle := \text{Pop}(Q)$ 6 if CONCRETIZE($\langle \varphi, i \rangle$) = $\top$ then 7 continue 8 if ISSAT($\mathcal{F}(\mathcal{O}_{i-1}) \wedge \varphi'$) then // The pob φ cannot be blocked at i 9 AddPredecessor($\langle \varphi, i \rangle$) 10 if ISSAT($R \wedge \text{Bad}$) then 11 return UNSAFE // Unsafe 12 else // The pob φ can be blocked at i 13 Block($\langle \varphi, i \rangle$) 14 for $0 \leq j \leq N$ do 15 for $\ell \in \mathcal{O}_j \setminus \mathcal{O}_{j+1}$ do 16 if $\mathcal{O}_j \wedge \text{Tr} \Rightarrow \ell'$ then 17 $\mathcal{O}_{j+1} := \mathcal{O}_{j+1} \wedge \ell$ // Propagate 18 if $\exists 0 \leq j < N \cdot \mathcal{O}_j \Rightarrow \mathcal{O}_{j-1}$ then 19 return $\langle \text{SAFE}, \mathcal{O}_j \rangle$ // Safe 20 if $\mathcal{O}_N \Rightarrow \neg \text{Bad}$ then 21 $N := N + 1$ // Unfold 22 Push($Q, \langle \text{Bad}, N \rangle$) </pre>	<pre> function ConcretizePOB: 23 $\langle \pi_1, \mathcal{L}_1 \rangle := \mathcal{C}_{pob}(\langle \varphi, i \rangle)$ 24 $\mathcal{L}_2 := \{ \ell \mid \ell \in \mathcal{L}_1 \wedge \text{ISSAT}(\ell \wedge \varphi) \wedge \text{ISSAT}(\neg \ell \wedge \varphi) \}$ 25 if $(\mathcal{L}_2 \neq \emptyset \wedge \text{NONLIN}(\pi_1) \wedge \text{ISSAT}(\bigwedge \mathcal{L}_2 \wedge \varphi))$ then 26 $\gamma := \text{CONCRETIZE}(\varphi, \langle \pi_1, \mathcal{L}_2 \rangle)$ 27 $k := \max\{j \mid \mathcal{O}_j \Rightarrow \neg \gamma\}$ 28 Push($Q, \langle \gamma, k \rangle$) // Concretize 29 Push($Q, \langle \varphi, i \rangle$) 30 return $\top$ 31 else return $\perp$ function AddPredecessor: 32 if ISSAT($\mathcal{F}(R) \wedge \varphi'$) then 33 find M_1 s.t. $M_1 \models \mathcal{F}(R) \wedge \varphi'$ 34 $s := (\text{MBP}(\bar{x}, \mathcal{F}(R), M_1)[\bar{x}' \mapsto \bar{x}])$ 35 $R := R \vee s$ // Successor 36 return 37 find M_2 s.t. $M_2 \models \Theta$ 38 $p := \text{MBP}(\bar{x}', \text{Tr} \wedge \varphi', M_2)$ 39 Push($Q, \langle p, i-1 \rangle$) // Predecessor 40 Push($Q, \langle \varphi, i \rangle$) function Block: 41 $\ell := \text{GEN}(\mathcal{F}(\mathcal{O}_{i-1}), \varphi')$ // Conflict 42 for $0 \leq j \leq i$ do $\mathcal{O}_j := \mathcal{O}_j \wedge \ell$ 43 $\langle \pi_3, \mathcal{L}_3 \rangle = \mathcal{C}_{lemma}(\ell)$ 44 $\alpha := \text{CONJECTURE}(\varphi, \mathcal{L}_3, R)$ 45 if $\alpha \neq \perp$ then 46 $k := \max\{j \mid \mathcal{O}_j \Rightarrow \neg \alpha\}$ 47 Push($Q, \langle \alpha, k \rangle$) // Conjecture 48 if $\neg \pi_3 = A \cdot \bar{x} \leq \bar{v}$ then 49 $\psi := \text{subsume}(\langle \pi_3, \mathcal{L}_3 \rangle)$ 50 $k := \max\{j \mid \mathcal{F}(\mathcal{O}_j) \Rightarrow \psi'\}$ 51 $\mathcal{O}_j := \mathcal{O}_j \wedge \psi$ for all $j \leq k+1$ // Subsume </pre>
---	--

2.3.5 Putting it all together

Having explained the implementation of the new rules for LIA, we now put all the ingredients together into an algorithm, GSPACER. In particular, we present our choices as to when to apply the new rules, and on which clusters of lemmas and POBs. As can be seen in Sec. 2.4, this implementation works very well on a wide range of benchmarks.

Alg. 7 presents GSPACER. The comments to the right side of a line refer to the

abstract rules in Alg. 1 and 4. Just like SPACER, GSPACER iteratively computes predecessors (line 9) and blocks them (line 13) in an infinite loop. Whenever a POB is proven to be reachable, the reachable states are updated (line 35). If *Bad* intersects with a reachable state, GSPACER terminates and returns UNSAFE (line 11). If one of the frames is an inductive invariant, GSPACER terminates with SAFE (line 19).

When a POB $\langle \varphi, i \rangle$ is handled, we first apply the **Concretize** rule, if possible (line 6). Recall that **CONCRETIZE** (Alg. 6) takes as input a cluster that partially blocks φ and has a non-linear pattern. To obtain such a cluster, we first find, using $\mathcal{C}_{pob}(\langle \varphi, i \rangle)$, a cluster $\langle \pi_1, \mathcal{L}_1 \rangle = \mathcal{C}_{O_k}(\pi_1)$, where $k \leq i$, that includes *some* lemma (from frame k) that blocks φ ; if none exists, $\mathcal{L}_1 = \emptyset$. We then filter out from $\mathcal{L}_1$ lemmas that completely block φ as well as lemmas that are irrelevant to φ , i.e., we obtain $\mathcal{L}_2$ by keeping only lemmas that partially block φ . We apply **CONCRETIZE** on $\langle \pi_1, \mathcal{L}_2 \rangle$ to obtain a new POB that under-approximates φ if (1) the remaining sub-cluster, $\mathcal{L}_2$, is non-empty, (2) the pattern, π_1 , is non-linear, and (3) $\bigwedge \mathcal{L}_2 \wedge \varphi$ is satisfiable, i.e., a part of φ is not blocked by any lemma in $\mathcal{L}_2$.

Once a POB is blocked, and a new lemma that blocks it, ℓ , is added to the frames, an attempt is made to apply the **Subsume** and **Conjecture** rules on a cluster that includes ℓ . To that end, the function $\mathcal{C}_{lemma}(\ell)$ finds a cluster $\langle \pi_3, \mathcal{L}_3 \rangle = \mathcal{C}_{O_i}(\pi_3)$ to which ℓ belongs (Sec. 2.3.1). Note that the choice of cluster is arbitrary. The rules are applied on $\langle \pi_3, \mathcal{L}_3 \rangle$ if the required pre-conditions are met (line 45 and line 49, respectively). When applicable, **SUBSUME** returns a new lemma that is added to the frames, while **CONJECTURE** returns a new POB that is added to the queue. Note that the latter is a *may* POB, in the sense that some of the states it represents *may not* lead to safety violation.

Ensuring progress. SPACER always makes progress: as its search continues, it establishes absence of counterexamples of deeper and deeper depths. However, GSPACER does not ensure progress. Specifically, unrestricted application of the **Concretize** and **Conjecture** rules can make GSPACER diverge even on executions of a fixed bound. In our implementation, we ensure progress by allotting a fixed amount of *gas* to each pattern, π , that forms a cluster. Each time **Concretize** or **Conjecture** is applied to a cluster with π as the pattern, π loses some gas. Whenever π runs out of gas, the rules are no longer applied to any cluster with π as the pattern. There are finitely many patterns (assuming LIA terms are normalized). Thus, in each bounded execution of GSPACER, the **Concretize** and **Conjecture** rules are applied only a finite number of times, thereby, ensuring progress. Since the **Subsume** rule does not hinder progress, it is applied without any restriction on gas.

Category	# Benchmarks
CHC-COMP18-LIA	438
CHC-COMP19-LIA-LIN	325
CHC-COMP19-LIA-nonLIN	283
CHC-COMP20-LIA-LIN	428
CHC-COMP20-LIA-nonLIN	416
CHC-COMP21-LIA-LIN	417
CHC-COMP21-LIA-nonLIN	432

Table 2.1: Number of LIA benchmarks in each category after removing duplicates.

Category	# Benchmarks
CHC-COMP18	132
CHC-COMP19	244
CHC-COMP20	499

Table 2.2: Number of LRA benchmarks in each category after removing duplicates.

2.3.6 Global Guidance for Linear Rational Arithmetic

So far, we have described our implementation of global guidance rules for LIA. The implementation for LRA is very similar. In fact, procedure for clustering lemmas, the pre-conditions for applying the three global guidance rules, and the procedures to conjecture and concretize POBs carry over without any modifications. The only difference is in the implementation of the *Subsume* rule (Alg. 5). For LIA, the procedure involves three main steps: computing convex hull (line 7), adding divisibility constraints (line 11), and eliminating auxiliary variables (line 15). Since we cannot have divisibility constraints for rational variables, we skip this step when implementing *Subsume* for LRA. The rest of the algorithm remains the same.

2.4 Evaluation

We implemented GSPACER (Alg. 7) as an extension to SPACER. The implementation has been merged with Z3¹. We use 10 as the starting value for gas for each pattern.

We used 3 sets of benchmarks for our evaluation: LIA and LRA instances from CHC-COMP 2018-21 [56] and benchmarks used in the evaluation of [106]. The CHC-COMP benchmarks have been curated from a variety of applications including Linux driver verification, synthesis and higher order program verification. The LIA benchmarks contain linear and non-linear CHCs whereas the LRA benchmarks only contain linear CHCs. We removed all duplicates from the benchmark suite. Since the LRA benchmarks in CHC-COMP 2021 are the same as those in CHC-COMP 2020,

¹Release version 4.11.2 <https://github.com/Z3Prover/z3/tree/z3-4.11.2>

Bench	GSPACER						SPACER						VBS	
	sc		bw		fw		sc		bw		fw			
	safe	unsafe	safe	unsafe	safe	unsafe	safe	unsafe	safe	unsafe	safe	unsafe	safe	unsafe
CHC18-Lia	208	55	210	55	205	52	115	61	149	64	142	62	223	73
CHC19-Lia-Lin	168	79	169	77	167	79	110	79	162	81	165	80	182	82
CHC19-Lia-NonLin	156	116	153	114	153	114	136	116	145	116	146	115	158	116
CHC20-Lia-Lin	223	133	227	132	224	134	190	135	209	136	209	135	230	136
CHC20-Lia-NonLin	203	200	205	203	204	201	196	203	202	203	201	202	207	203
CHC21-Lia-Lin	198	87	199	90	200	88	141	90	175	92	177	92	205	95
CHC21-Lia-NonLin	251	134	254	129	251	127	225	138	236	134	230	133	262	139
Total	1407	804	1417	800	1404	795	1113	822	1278	826	1270	819	1467	844

Table 2.3: Comparison between SPACER and GSPACER on LIA benchmarks from CHC-COMP.

we omit them in our evaluation. The numbers of instances in each category, after removing duplicates, are shown in Tab. 2.1 and Tab. 2.2.

To evaluate our implementation, we have conducted 3 sets of experiments. All experiments were run on Intel E5-2690 V2 CPU at 3GHz with 128GB memory with a timeout of 10 minutes. The first set of experiments (Sec. 2.4.1) evaluate the performance of local reasoning with global guidance against pure local reasoning. The second set of experiments (Sec. 2.4.2) compare the performance of local reasoning with global guidance to solely global reasoning. The third set of experiments (Sec. 2.4.3) compare the effectiveness of each of the global guidance rules.

We use SPACER as a *baseline* for pure local reasoning because it dominated CHC-COMP by solving 85% of the LIA benchmarks in CHC-COMP 2019 (20% more than the runner up) and 60% of the LIA benchmarks in CHC-COMP 2018 (10% more than runner up). In CHC-COMP 2020 and CHC-COMP 2021, GSPACER participated instead of SPACER and won all LIA tracks. Our evaluation shows that GSPACER outperforms SPACER both in terms of number of solved instances and, more importantly, in overall robustness.

We used the ML-based data-driven invariant inference tool LINEARARBITRARY [106] as a baseline for purely global reasoning. Compared to other similar approaches, LINEARARBITRARY stands out by supporting invariants with arbitrary Boolean structure over arbitrary linear predicates. It is completely automated and does not require user-provided predicates, grammars, or any other guidance. For the comparison with LINEARARBITRARY, we have used both the CHC-COMP benchmarks, as well as the benchmarks from the artifact evaluation of [106]. The machine and timeout remain the same. Our evaluation shows that GSPACER is superior in this case as well.

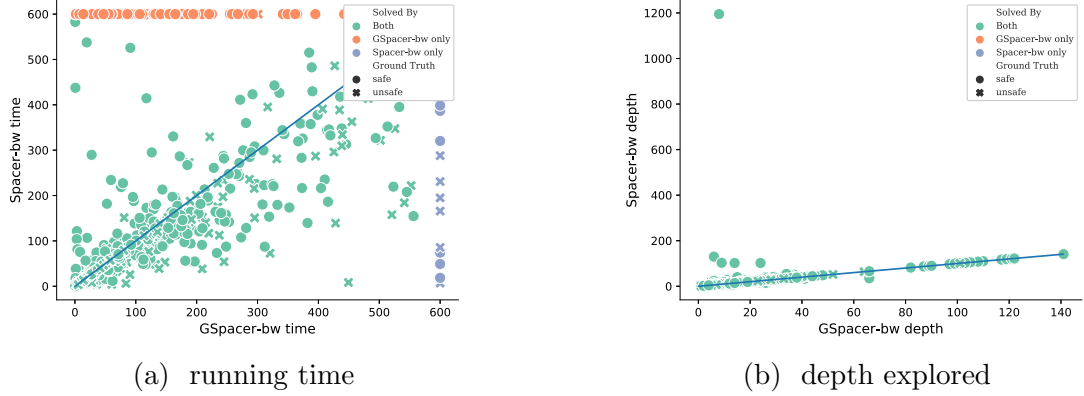


Figure 2.2: Best configurations on LIA benchmarks: GSPACER versus SPACER.

2.4.1 Comparison with Spacer.

Comparison on LIA instances Tab. 2.3 summarizes the comparison between SPACER and GSPACER on LIA instances from CHC-COMP. Since both tools can use a variety of interpolation strategies during lemma generalization (Line 41 in Alg. 7), we compare three different configurations of each: *bw* and *fw* stand for two interpolation strategies, *backward* and *forward*, respectively, already implemented in SPACER, and *sc* stands for turning interpolation off and generalizing lemmas only by *subset clauses* computed by inductive generalization. VBS stands for the Virtual Best Solver.

Any configuration of GSPACER solves significantly more instances than even the best configuration of SPACER. Fig. 2.2 provides a more detailed comparison between the best configurations of both tools in terms of running time and depth of convergence. There is no clear trend in terms of running time on instances solved by both tools. This is not surprising — SMT-solving run time is highly non-deterministic and any change in strategy has a significant impact on performance of SMT queries involved. In terms of depth, it is clear that GSPACER converges at the same or lower depth. The depth is significantly lower for instances solved only by GSPACER.

Moreover, the performance of GSPACER is not significantly affected by the interpolation strategy used. In fact, the configuration *sc* in which interpolation is disabled performs the best in non-linear instances from CHC-COMP 2019, and only slightly worse in other categories. In comparison, disabling interpolation hurts SPACER significantly.

Fig. 2.3 provides a detailed comparison of GSPACER with and without interpolation. Interpolation makes no difference to the depth of convergence. This implies that lemmas that are discovered by interpolation are discovered as efficiently by the global

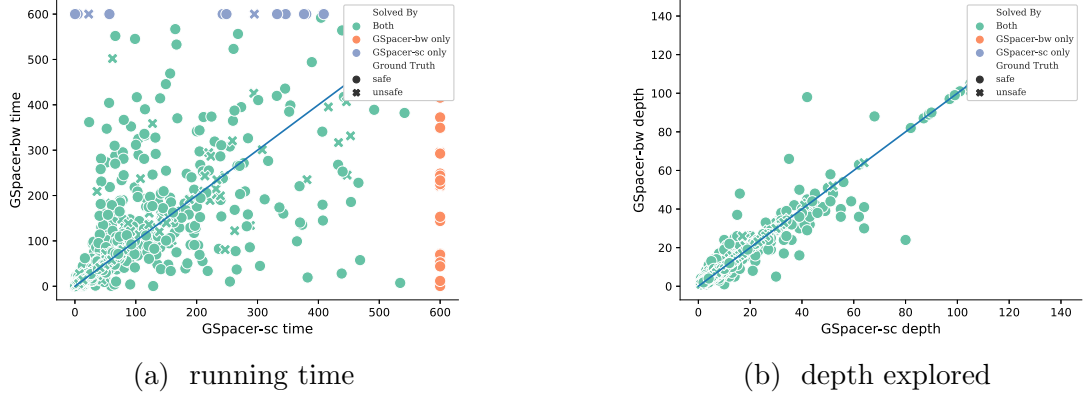


Figure 2.3: Comparing GSPACER with different interpolation tactics on LIA benchmarks.

Bench	GSPACER						SPACER						VBS	
	sc		bw		fw		sc		bw		fw		safe	unsafe
CHC18	55	8	51	7	57	7	56	8	54	8	55	8	62	8
CHC19	110	47	112	45	113	45	78	47	101	46	101	46	123	47
CHC20	215	73	214	74	210	74	186	75	199	74	207	74	239	75
Total	380	128	377	126	380	126	320	130	354	128	363	128	424	130

Table 2.4: Comparison between SPACER and GSPACER on LRA benchmarks from CHC-COMP.

rules of GSPACER. On the other hand, interpolation significantly increases the running time. Interestingly, the time spent in interpolation itself is insignificant. However, the lemmas produced by interpolation tend to slow down other aspects of the algorithm. Most of the slow down is in increased time for inductive generalization and in computation of predecessors. The comparison between the other interpolation-enabled strategy and GSPACER-sc shows a similar trend.

Comparison on LRA benchmarks To see how well the global guidance rules scale across theories, we compared GSPACER with SPACER on LRA benchmarks from CHC-COMP. The results are summarized in Tab. 2.4. The meanings of the columns are the same as in Tab. 2.3.

Most significantly, we find that global guidance is robust. While SPACER has a noticeable drop in performance when interpolation is turned off, GSPACER does not.

Second, while the number of instances solved by GSPACER is similar to the number of instances solved by SPACER, the instances solved are different. In particular, the VBS significantly outperforms both solvers. Finally, the improvement in GSPACER on LRA benchmarks is not as significant as on LIA benchmarks. We believe this is a property of the benchmark set rather than the intrinsic difference between LIA and LRA. We conjecture that additional global guidance rules, specifically ones taking advantage of symmetry in the lemmas, will boost the performance of GSPACER.

Comparison on unsafe instances SPACER consistently solves more unsafe instances than GSPACER. This is an expected side effect of the global guidance rules. GSPACER explores different ways to find an inductive invariant and, on unsafe instances, this exploration is completely wasteful. This exploration comes at a cost. Each POB that global guidance generates is checked for reachability and, if blocked, generalized using inductive generalization. Both these are one or more queries to an SMT solver. The cost of exploration is checked using gas (Sec. 2.3.5). For each cluster, gas is initially set to a constant value and decreased each time a global guidance rule is applied. Once a cluster runs out of gas, global guidance rules are not applied to the cluster. While this heuristic reduces exploration, it does not reduce it to zero, hence the deterioration in performance.

2.4.2 Comparison with LinearArbitrary.

In [106], the authors show that LINEARARBITRARY, to which we refer as LARB for short, significantly outperforms SPACER on a curated subset of benchmarks from SV-COMP [24] competition.

We first compare LARB against GSPACER on CHC-COMP instances. The first two rows of Tab. 2.5 shows the number of instances solved by both solvers. We see that LARB solves way fewer instances than even baseline SPACER.

For a more meaningful comparison, we compared SPACER, LARB, and GSPACER on the benchmarks from the artifact evaluation of [106]. The results are summarized in the last row of Tab. 2.5. As expected, LARB outperforms the baseline SPACER on the safe benchmarks. On unsafe benchmarks, SPACER is significantly better than LARB. In both categories, GSPACER dominates solving more safe benchmarks than either SPACER or LARB, while matching performance of SPACER on unsafe instances. Furthermore, GSPACER remains orders of magnitude faster than LARB on benchmarks that are solved by both. This comparison shows that incorporating local reasoning with global guidance not only mitigates its shortcomings but also surpasses global data-driven reasoning.

Bench	SPACER		LARb		GSPACER	
	safe	unsafe	safe	unsafe	safe	unsafe
CHC-18-LIA	149	64	64	8	210	55
CHC-19-LIA-LIN	162	81	131	18	169	77
PLDI18	215	68	270	65	280	67

Table 2.5: Comparison with LARb using CHC-COMP benchmarks.

Bench	Conjecture		Concretize		Subsume		VBS	
	safe	unsafe	safe	unsafe	safe	unsafe	safe	unsafe
CHC18-Lia	146	52	149	54	191	51	208	56
CHC19-Lia-Lin	160	77	161	76	165	79	168	79
CHC19-Lia-NonLin	144	114	144	115	152	114	154	115
CHC20-Lia-Lin	209	132	212	133	222	134	226	134
CHC20-Lia-NonLin	201	201	201	200	204	202	205	202
CHC21-Lia-Lin	172	88	174	87	193	88	200	89
CHC21-Lia-NonLin	234	132	233	132	250	130	254	132
Total	1266	796	1274	797	1377	798	1415	807

Table 2.6: Effectiveness of individual rules of GSPACER on CHC-COMP LIA instances.

2.4.3 Effectiveness of individual Global Guidance rules

Tab. 2.6 summarizes the effectiveness of individual global guidance rules on CHC-COMP LIA benchmarks. Each column in Tab. 2.6 corresponds to running GSPACER with just one of the three global guidance rules enabled. We see that the Subsume configuration (that is, running GSPACER with just the Subsume rule) solves many more instances than either of the other configurations. However, even the Subsume rule individually does not solve merely as many instances as GSPACER with all three rules combined. Fig. 2.4 compares the running time and depth of convergence of Subsume configuration (x-axis) against those of the full GSPACER (y-axis). Clearly, there are many instances in which GSPACER with all three rules converges at a significantly lower depth than the GSPACER with just the Subsume rule.

2.5 Related Work

The limitations of local reasoning in SMT-based infinite state model checking are well known. Most commonly, they are addressed with either (a) different strategies for local generalization in interpolation (e.g., [1], [16], [77], [95]), or (b) shifting the focus

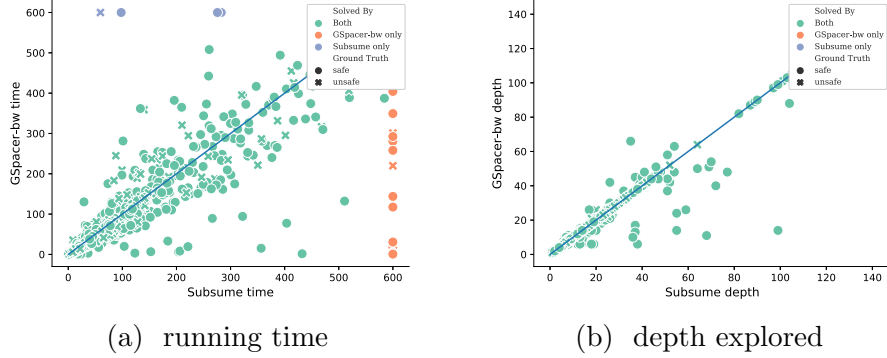


Figure 2.4: Comparing GSPACER with just the **Subsume** rule (x-axis) against GSPACER with all three rules together (y-axis).

to *global* invariant inference by learning an invariant of a restricted shape (e.g., [19], [37], [41], [47], [106]).

Interpolation strategies. The CSMT algorithm [1] attempts to find simple interpolants (i.e., interpolants with fewer literals), arguing that simple interpolants are more likely to generalize. Given two formulas as disjunctions of halfspaces, CSMT samples a subset of halfspaces from both formulas and computes a single halfspace separating the sets of disjuncts using an algorithm called **HALFITP**. **HALFITP** is similar to the subsume rule: both techniques attempt to learn a single lemma to summarize a set of lemmas. However, the techniques are quite different. **HALFITP** uses an SMT solver to find Farkas coefficients for the two sets of lemmas whereas subsume leverages syntactic convex closure. Therefore, when the sets of lemmas cannot be separated using a single halfspace, **HALFITP** fails but subsume still succeeds. When **HALFITP** fails to find a single halfspace separating two samples, CSMT generates more samples by merging and splitting existing samples, applies **HALFITP** on them, and combines the resulting halfspaces into a full interpolant. The samples (formed by merging and splitting) are a key factor in deciding the final interpolant. CSMT is semantic: sampling is done based on the models produced by an SMT solver. In contrast, clustering in global guidance is purely syntactic. They behave quite differently under different circumstances. For example, in the presence of outliers, CSMT learns a disjunction of conjunctions as an interpolant for the whole samples, including the outliers. Whereas, global guidance attempts to cluster the sets of lemmas in multiple ways, sometimes including the outliers and other times not. For each such cluster, it employs some guidance rules (not necessarily the same ones) to learn lemmas. In this way, global guidance learns multiple lemmas, some of which might be generalizable. On the other hand, if each of

the lemmas in a cluster contain too many literals that are syntactically different yet semantically the same, global guidance does not have a heuristic to identify the best way to cluster them. Whereas, CSMT is unaffected by syntactic differences between lemmas. Another main difference between the algorithms is that global guidance rules combine to mitigate effects of wrong generalizations. If the same guidance rule is applied several times on similar clusters to produce similar lemmas, these lemmas form a cluster and some other guidance rule produces a generalization for the new cluster. Finally, global guidance corrects both myopic generalization (Fig. 2.1a) and excessive generalization (Fig. 2.1b) in the sample set whereas CSMT cannot correct excessive generalization.

On the other extreme, Blichia et al. [16] decompose interpolants to be numerically simpler, but with more literals. This is similar to our concretize rule where we decouple terms that form a non-linear pattern. However, the decomposition is local: it is guided by dependence between Farkas coefficients and is not affected by how the model checking algorithm progresses. It might help with some cases of excessive generalization but not with myopic generalization. Deciding *locally* between CSMT [1] and the algorithm presented by Blichia et al. [16] or on their combination (i.e., some parts of an interpolant might need to be split while others combined) seems difficult. Schindler and Jovanović [95] propose local interpolation that bounds the number of lemmas generated from a single POB (which helps with Fig. 2.1c), but only if inductive generalization is disabled. Finally, [77] suggests using external guidance, in a form of predicates or terms, to guide interpolation. In contrast, GSPACER uses global guidance, based on the current proof, to direct different local generalization strategies. Thus, the guidance is automatically tuned to the specific instance at hand rather than to a domain of problems.

Global invariant inference. An alternative to inferring lemmas for the inductive invariant by blocking counterexamples is to enumerate the space of potential candidate invariants [19], [37], [41], [47], [106]. This does not suffer from the pitfall of local reasoning. However, it is only effective when the search space is constrained. While these approaches perform well on their target domain, they do not generalize well to a diverse set of benchmarks, as illustrated by results of CHC-COMP and our empirical evaluation in Sec. 2.4.

Locality in SMT and IMC. Local reasoning is also a known issue in SMT, and, in particular, in DPLL(T) (e.g., [82]). SMT solvers are to Model checkers what theory solvers are to SMT solvers.

However, we are not aware of global guidance techniques for SMT solvers. Interpolation-based Model Checking (IMC) [80], [81] that uses interpolants from proofs, inherits

the problem. Compared to IMC, the propagation phase and inductive generalization of IC3 [17], can be seen as providing global guidance using lemmas found in other parts of the search-space. In contrast, GSPACER magnifies such global guidance by exploiting patterns within the lemmas themselves.

IC3-SMT-based Model Checkers. There are a number of IC3-style SMT-based infinite state model checkers, including [23], [64], [71]. To our knowledge, none extend the IC3-SMT framework with a global guidance. A rule similar to **Subsume** is suggested in [102] for the theory of bit-vectors and in [13] for LRA, but in both cases without global guidance. In [13], it is implemented via a combination of syntactic closure with interpolation, whereas we use MBP instead of interpolation. Refinement State Mining in [12] uses similar insights to our **Subsume** rule to refine predicate abstraction.

2.6 Conclusion

This chapter introduces *global guidance* to mitigate the limitations of the local reasoning performed by SMT-based IC3-style model checking algorithms. Global guidance is necessary to redirect such algorithms from divergence due to persistent local reasoning. To this end, we present three general rules that introduce new lemmas and POBs by taking a global view of the lemmas learned so far. The new rules are not theory-specific, and, as demonstrated by Alg. 7, can be incorporated to IC3-style solvers without modifying existing architecture. We instantiate, and implement, the rules for LIA and LRA in GSPACER, which extends SPACER.

Our evaluation shows that global guidance brings significant improvements to local reasoning, and surpasses invariant inference based solely on global reasoning. More importantly, global guidance decouples SPACER’s dependency on interpolation strategy and performs almost equally well under all three interpolation schemes we consider. As such, using global guidance in the context of theories for which no good interpolation procedure exists arises as a promising direction for future research. For example, in the theory of Fixed-Size Bit-Vectors [65].

Chapter 3

CHC Solvers For Programs Manipulating Data Types

Recursive data structures, such as Lists and Trees are ubiquitous in programming. Proofs of correctness of programs that use such data structures rely on the theory of Algebraic Data Types (ADT) – the logic counter-part to data structures, and on Recursively Defined Functions (RDF), to represent properties (or abstractions) of data types. For example, length of a List, height of a Tree, are captured by RDFs.

As an example, consider the imperative Dafny [75] program in Fig. 3.1. The program manipulates a list represented by the variable `y`. Note that both the specification (in `requires`) and an inductive invariant (in `invariant`) require an RDF `length` (also shown in the figure). In this case, the RDF `length` appears both in the specification and the invariant, but that need not be the case. Similarly, programs without RDFs might require RDFs in their invariants (for example, via ghost code).

Since the satisfiability of the combination of ADT and RDF is undecidable, even validating a candidate inductive invariant is difficult. While Dafny [75], and similar deductive verification tools (e.g., Why3 [39], Viper [86]), provide ample support for reasoning with ADTs and RDFs, the situation is much worse in automated verification, where inductive invariants need to be synthesized and not just verified.

In the case of programs with data structures, such as the one in Fig. 3.1, the constraints of the CHCs are over a combination of theories including Linear Integer Arithmetic (LIA) (to model `i`), ADTs (to model `y`), and RDFs (to model `length`). Supporting the combination of such theories in a CHC-solver is complicated because of the undecidability of even the most basic decision problems in this case.

In fact, several existing solvers, including SPACER, already have some support for CHC modulo ADTs [15]. However, as we show above, reasoning about ADTs almost always requires reasoning with RDFs as well, and RDFs are not supported.

```

datatype List = Nil | Insert(head: int, tail: List)

method main(y: List, i: int)
  requires length(y) == i;
  { _y, _i := y, i;
    while (*)
      invariant _i == length(_y)
      { if (_y != Nil)
        { _y, _i := _y.tail, _i - 1; }}
    assert (_i >= 0); }

function length(xs: List): int {
  match xs
  case Nil => 0
  case Insert(h, t) =>
    1 + length(t)
}

```

Figure 3.1: A Dafny program that manipulates a list.

It is reasonable to view the relationship of RDFs to ADTs as the relationship between quantifiers and arrays – reasoning about array manipulating programs requires quantifiers for the same reasons as RDFs are needed for ADT-manipulating programs.

The key to our approach is a dual treatment of RDFs in CHCs C . On one hand, we approximate an axiom defining an RDF in C by CHCs without RDFs, through a process called *relationification*, resulting in an *equisatisfiable* CHC C_F . That is, C_F preserves both proofs (satisfiability) and counterexamples (unsatisfiability). However, it does not preserve solutions — expressing a solution for C_F might require a richer logic than the corresponding solution for C , since the former not only needs to capture an inductive invariant for the programs, but also needs to summarize the relationified RDF.

On the other hand, we approximate RDFs in C by two novel abstractions that inject into C a bounded unfolding of the RDF: both abstractions apply a finite unfolding of the recursive definition, after which the *k-instantiation* abstraction treats remaining RDF applications as uninterpreted functions, while the *uninterpreted function* abstraction omits them. In both abstractions, RDFs are replaced by their partial definition — thus, gaining decidability of basic queries. Relationification enables CHC solvers to deduce inductive summaries of RDFs whereas the abstractions provide unfoldings of RDFs. Inductive summaries and unfoldings of RDFs allow us to solve RDF constraints without facing the undecidability issues introduced by RDFs. Both abstractions ensure decidability of the underlying theory.

The combination of relationification with each of the abstractions (*k*-instantiation or uninterpreted functions), parameterized by an unfolding bound k , results in two systems of abstract CHCs. Each system is an over-approximation (because of the abstraction). Nonetheless, relationification ensures that the abstract CHCs are *equisatisfiable* to the original ones. The *k*-instantiation abstraction is perfect for finding solutions, with the number of solutions preserved by the abstraction determined by the value of k . The uninterpreted function abstraction is a CHC over ADTs only and

is perfect for showing unsatisfiability (i.e., finding counterexamples). The abstractions, parameterized by k , thus, create a hierarchy of abstractions (shown in Fig. 3.4).

We present an algorithm, RACER, that operates in the space of the hierarchy of k -instantiation abstractions, dynamically adjusting k , while simultaneously searching through the counterexamples in the uninterpreted function abstraction. While this sounds complex, it fits well into an IC3-style framework that iteratively explores bounded unfoldings of the system. Remarkably, this integration seamlessly combines learning inductive properties of RDFs with learning inductive invariants of the system, while exploiting the SMT support for non-inductive RDF reasoning via incremental unfolding of RDFs.

An important characteristic of our algorithm is that it is guaranteed to make progress and to discover counterexamples, if they exist. The problem of CHC solving is more general than SMT-solving (i.e., inferring invariants vs. checking invariants). Even CHC modulo LIA is undecidable, thus, there is little hope for completeness results when considering CHC solving. On the other hand, unsatisfiability of CHCs (and FOL, in general) is recursively enumerable. For CHCs, unsatisfiability corresponds to the existence of a counterexample to safety. As a result, progress — which implies, in particular, discovery of counterexamples — is key.

The integration of relationification with abstractions of RDFs is crucial for ensuring progress while preserving solutions of the CHCs. While k -deep unfolding alone does not suffice for making progress, relationification ensures progress as it allows to learn bounded properties of RDFs. On the other hand, relationification alone does not preserve solutions of the CHCs due to the inability to express summaries of RDFs. Thus, the combination of relationification with unfolding of RDFs is critical for maintaining progress while preserving (many) solutions. It allows to combine inference of inductive summaries of RDFs, which are needed to establish validity of solutions, with some unfoldings, guided by the property.

We develop a single procedure for solving CHCs with ADTs, RDFs, and LIA, while maintaining progress, and without affecting performance on LIA. This is in contrast to other related work, which either automates induction assuming that arithmetic is axiomatized [92], [94] or does not admit RDFs [38], [61], [71].

In our exposition, we restrict RDFs to catamorphisms (a.k.a. generalized folds). The RDF `length` in Fig. 3.1 is an example. Extending our ideas to other well defined RDFs is explained in Sec. 3.3.

Limitations Our current approach is limited in two ways. First, we assume that the RDFs are specified. That is, we only synthesize inductive invariants, but not the supporting functions. Second, we do not synthesize new applications of RDFs, relying

on a form of term abstraction [2]. We leave addressing these orthogonal problems to future work.

In this chapter, we explain a new technique to solve CHCs modulo ADTs and RDFs. In particular, we describe (1) a new hierarchy of abstractions for CHC modulo ADT and RDF that combine relationification with bounded unfoldings of RDFs, thus, preserving counterexamples and (many) solutions; (2) an IC3-based procedure RACER for solving CHC modulo ADT and RDF that searches through the hierarchy of abstractions; It integrates into and extends the SPACER framework, and provides progress guarantees; (3) an empirical evaluation on a variety of benchmarks.

This chapter has been published in the 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2022) [67].

3.1 Overview

In this section we give an overview of our approach using a motivating example. Our focus is automatic safety verification of imperative programs that manipulate Algebraic Data Types (ADTs), such as lists, trees, or queues. Such programs and their specifications often require Recursively Defined Functions (RDFs), such as *length*, *size*, etc. Thus, we must support reasoning about the combination of ADTs and RDFs.

The key challenge of automated safety verification is automatically synthesizing and validating inductive invariants that summarize the behavior of all the loops. In case of ADTs, expressing invariants often requires RDFs, and, sometimes, RDFs that are not even present in the original program. For example, inductive invariants of list-manipulating programs often require to use the length of a list (and other recursive functions that compute inductive properties of lists). Thus, automatic reasoning over ADTs requires (a) synthesis of inductive invariants over ADTs and RDFs, and (b) synthesis of useful RDFs. In this chapter, we focus on the first challenge – invariant synthesis. We assume that RDFs and their applications to program variables are either heuristically extracted from the program or are provided by the users. Synthesizing useful RDFs can be done as an abstraction-refinement loop over our work.

Motivating Example To illustrate our approach, consider the list manipulating program in Fig. 3.1. `List` is a list ADT with two constructors `Nil` and `Insert`. `length` is an RDF that computes the length of a list. The method `main` takes as input a list `y` and its length `i`. It removes elements from the list one by one in a loop, decreasing `i` after each removal. Whenever the loop exits, it is asserted that the variable `i` has a non negative value. The program is safe, as witnessed by the following inductive invariant `i == length(y)`. While this program is simple, automatically verifying

$$\begin{aligned}
& \text{length}(y) \approx i \Rightarrow \text{Inv}(y, i, i) \\
& \text{Inv}(y, i, j) \wedge \text{length}(y) \approx j \wedge y \not\approx \text{nil} \wedge y' \approx \text{tail}(y) \wedge \text{length}(y') \approx j' \Rightarrow \text{Inv}(y', i-1, j') \\
& \text{Inv}(y, i, j) \wedge \text{length}(y) \approx j \wedge i < 0 \Rightarrow \perp
\end{aligned}$$

(a) CHC modulo ADT and RDF encoding safety of the program from Fig. 3.1.

$$\begin{aligned}
& y \approx \text{nil} \wedge z \approx 0 \Rightarrow \text{Length}(y, z) \\
& y \not\approx \text{nil} \wedge \text{Length}(\text{tail}(y), l) \wedge z \approx 1 + l \Rightarrow \text{Length}(y, z) \\
& \text{Length}(y, i) \Rightarrow \text{Inv}(y, i, i) \\
& \text{Inv}(y, i, j) \wedge \text{Length}(y, j) \wedge y \not\approx \text{nil} \wedge y' \approx \text{tail}(y) \wedge \text{Length}(y', j') \Rightarrow \text{Inv}(y', i-1, j') \\
& \text{Inv}(y, i, j) \wedge \text{Length}(y, j) \wedge i < 0 \Rightarrow \perp
\end{aligned}$$

(b) CHCs C_F obtained by relationifying the RDFs in Fig. 3.2a

$$\begin{aligned}
& y \approx \text{nil} \wedge z \approx 0 \Rightarrow \text{Length}(y, z) \\
& y \not\approx \text{nil} \wedge \text{Length}(\text{tail}(y), l) \wedge z \approx 1 + l \Rightarrow \text{Length}(y, z) \\
& \text{Length}(y, i) \wedge i \approx \text{length}(y) \Rightarrow \text{Inv}(y, i, i) \\
& \text{Length}(y, j) \wedge j \approx \text{length}(y) \wedge \text{Length}(y', j') \wedge j' \approx \text{length}(y') \wedge \\
& \quad \text{Inv}(y, i, j) \wedge y \not\approx \text{nil} \wedge y' \approx \text{tail}(y) \Rightarrow \text{Inv}(y', i-1, j') \\
& \text{Length}(y, j) \wedge j \approx \text{length}(y) \wedge \text{Inv}(y, i, j) \wedge i < 0 \Rightarrow \perp
\end{aligned}$$

(c) C_{fF} : the result of adding back the *length* RDF to relationified CHCs C_F from Fig. 3.2b.

Figure 3.2: Example CHCs for (a) encoding of Dafny program from Fig. 3.1; (b) relationification, (c) relationifications with RDFs. Modifications and additions are highlighted in yellow and green, respectively.

its safety is difficult for many existing verification techniques. Specifically, there are multiple theories involved that require to simultaneously reason about integer variables (i.e., variable *i*), ADT variables (i.e., variable *y*), and RDFs (i.e., function *length*). Furthermore, the proof that the suggested invariant is inductive requires guessing an inductive property of *length*: *length*(*y*) is non-negative for any list *y*.

A CHC encoding of our example program from Fig. 3.1 is shown in Fig. 3.2a. In this encoding, the uninterpreted predicate *Inv* summarizes all behaviours of the

$$\begin{aligned}
& y \approx \text{nil} \wedge z \approx 0 \Rightarrow \text{Length}(y, z) \\
& y \not\approx \text{nil} \wedge \text{Length}(\text{tail}(y), l) \wedge z \approx 1 + l \Rightarrow \text{Length}(y, z) \\
& \text{Length}(y, i) \wedge i \approx \text{length}_{uf}(y) \wedge i \approx \text{ite}(y \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(y))) \Rightarrow \text{Inv}(y, i, i) \\
& \text{Length}(y, j) \wedge j \approx \text{length}_{uf}(y) \wedge j \approx \text{ite}(y \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(y))) \wedge \\
& \text{Length}(y', j') \wedge j' \approx \text{length}_{uf}(y') \wedge j' \approx \text{ite}(y' \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(y'))) \wedge \\
& \text{Inv}(y, i, j) \wedge y \not\approx \text{nil} \wedge y' \approx \text{tail}(y) \wedge i' \approx i - 1 \Rightarrow \\
& \text{Inv}(y', i', j') \\
& \text{Length}(y, j) \wedge j \approx \text{length}_{uf}(y) \wedge j \approx \text{ite}(y \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(y))) \wedge \\
& \text{Inv}(y, i, j) \wedge i < 0 \Rightarrow \perp
\end{aligned}$$

(a) C_{fF}^1 : the 1-instantiation abstraction of CHCs C_{fF} from Fig. 3.2c. length_{uf} is an uninterpreted function.

$$\begin{aligned}
& y \approx \text{nil} \wedge z \approx 0 \Rightarrow \text{Length}(y, z) \\
& y \not\approx \text{nil} \wedge \text{Length}(\text{tail}(y), l) \wedge z \approx 1 + l \Rightarrow \text{Length}(y, z) \\
& \text{Length}(y, i) \wedge i \approx \text{ite}(y \approx \text{nil}, 0, 1 + w) \Rightarrow \text{Inv}(y, i, i) \\
& \text{Length}(y, j) \wedge j \approx \text{ite}(y \approx \text{nil}, 0, 1 + w) \wedge \\
& \text{Length}(y', j') \wedge j' \approx \text{ite}(y' \approx \text{nil}, 0, 1 + w') \wedge \\
& \text{Inv}(y, i, j) \wedge y \not\approx \text{nil} \wedge y' \approx \text{tail}(y) \wedge i' \approx i - 1 \Rightarrow \\
& \text{Inv}(y', i', j') \\
& \text{Length}(y, j) \wedge j \approx \text{ite}(y \approx \text{nil}, 0, 1 + w) \wedge \text{Inv}(y, i, j) \wedge i < 0 \Rightarrow \perp
\end{aligned}$$

(b) $C_{fF}^{k\uparrow}$: Uninterpreted functions abstraction of CHCs C_{fF} from Fig. 3.2c.

Figure 3.3: Examples of (a) k -instantiation and (b) uninterpreted function abstractions of CHCs from Fig. 3.2c.

`main` method. Essentially, it represents a desired inductive invariant of the loop. Inv takes three arguments: the first two are the same as those of `main`, and the third encodes the length of the list y . The third argument is required so that $\text{length}(y)$ can appear in an inductive invariant. In practice, such terms are obtained by greedy term abstraction, as we explain in Sec. 3.7.

A valid solution to the CHC encoding (Fig. 3.2a) assigns to predicate $\text{Inv}(y, i, j)$, the formula $(i \approx j)$, where j encodes the length of list y , as captured by the constraint $\text{length}(y) \approx j$. Hence, this is essentially the inductive invariant $i == \text{length}(y)$. Representing the RDF application $\text{length}(y)$ by j follows the idea of *term abstraction*.

It allows us to separate the task of synthesizing the required RDF applications from the task of inferring inductive invariants over them, which is the focus of the chapter. Namely, instead of synthesizing RDF applications in the process of inferring an invariant (solution), we infer invariants over variables that represent a predefined set of RDF applications. With this encoding, deciding whether the original program is safe is reduced to inferring solutions to the corresponding CHC encoding of its verification condition. We stress that inferring solutions (i.e., inductive invariants) is much harder than checking that a candidate invariant is inductive, as traditionally done in deductive verification.

Challenges for Solving CHCs modulo RDFs As we have seen, solving CHCs is a challenging problem in general, and has received a lot of attention. CHCs modulo ADTs and RDFs bare additional challenges:

Undecidability of validating solutions In the presence of RDFs, determining validity of FOL formulas is undecidable. Thus, even checking whether a candidate formula is an inductive invariant (a step used by many existing verification techniques) is undecidable. Validity may depend on properties of RDFs that must be established inductively (i.e., no amount of finite unfolding of the recursive definition is sufficient). Applying induction requires coming up with an inductive hypothesis, which, intuitively, is a source of undecidability.

The undecidability challenge significantly limits applicability of guess-and-check techniques, such as Houdini [40], [41], in this context. This extends to all guess-and-check techniques including more recent approaches based on machine learning [47] and grammar-based synthesis [38]. One exception is the CVC4 SMT solver that combines candidate enumeration with induction [92]. However, the technique in [92] is not fully automated (but rather user or problem guided). It assumes that required inductive hypotheses are either part of the input, or are provided by the user as sub-goals. This is justified by their application domain (automating deductive verification), but is not very effective in an automated verification setting, as recently shown in [28]. There are also known sub-classes of RDFs for which validity is decidable. One such class of *infinitely surjective catamorphisms (ISCs)* over ADTs is presented by Suter et al. [96]. However, it is not clear that determining that an RDF is of an appropriate class (i.e., an ISC) is decidable.

Incompleteness of RDF elimination As RDFs make validating solutions undecidable, a natural direction of attack is to reduce the problem by somehow eliminating RDFs, while preserving the key characteristics of the verification task. One approach is *relationification*: encoding RDFs as CHC predicates, constrained by additional Horn

clauses. For example, relationification of the CHCs in Fig. 3.2a gives us the CHCs in Fig. 3.2b. Relationified CHCs do not contain RDFs. Instead, they are over ADTs (and other background theories). Therefore, checking the validity of solutions is decidable. Intuitively, relationification replaces recursive functions by their implementation. One of our results is to show that while relationification preserves satisfiability, it loses solutions over the language of CHCs. Roughly, this is because a solution to the CHCs should now also include a summary of the relationified RDF. However, there are cases where such a summary cannot be specified without RDFs (see Ex. 7).

Our approach We present a new approach for solving CHCs modulo ADT and RDF that addresses the above challenges. The core of our approach is the *combination* of relationified predicates with a novel *abstraction* of RDFs based on finite RDF unfoldings.

The use of (abstraction of) RDFs allows to express solutions that cannot be expressed without RDFs, thus, tackling the second challenge. Abstractions are key to ensuring that checking validity of solutions remains decidable (tackling the first challenge). Unfortunately, abstractions may not be sufficient to validate solutions, specifically, in the cases where finite unfoldings do not suffice. This is where the interplay with the relationified predicates, that are conjoined with the RDF abstractions, comes into play: relationified predicates allow to use IC3/PDR (see Sec. 1.6) style algorithms to automatically generate necessary inductive hypotheses and lemmas that are needed in order to validate solutions. However, this is not the only role of the relationified predicates. The Horn clauses that encode the relationified predicates ensure that despite the RDF abstraction, unsatisfiability is preserved. Namely, if the original CHC is unsatisfiable, so is the transformed one. Indeed, a key characteristic of our approach is that it is *refutationally complete* — whenever the (original) CHCs are unsatisfiable (i.e., the program is unsafe), our approach is guaranteed to terminate with a finite counterexample. In fact, our approach has a stronger property from which the former follows: the approach is ensured to make progress in verifying *bounded safety* for increasing bounds.

Relationification side by side with RDFs The first step in our approach is to extend a given set of CHCs modulo RDFs with relationification of the RDFs. That is, we conjoin the CHCs with clauses that constrain the relationified predicates and conjoin each RDF application with an application of the relationified predicate. For example, the result of this transformation on CHCs in Fig. 3.2a is shown in Fig. 3.2c. The newly added clauses, and newly added predicate applications are highlighted in green and yellow, respectively. The resulting CHCs preserve both solutions of the original CHCs (when they are satisfiable), and unsatisfiability (when the original

CHCs are unsatisfiable).

Abstractions of RDFs The second step in our approach is a new IC3-style algorithm, called RACER, that employs two orthogonal abstractions simultaneously in order to solve the augmented CHCs. These abstractions, called *k-instantiation* and *uninterpreted functions* abstraction, respectively, are key to avoiding the undecidability barrier (for solution validation).

The *k*-instantiation abstraction unfolds all occurrences of RDFs in a formula *k* times and replaces any remaining RDFs with uninterpreted functions (UF). Applying the *k*-instantiation abstraction to CHCs modulo ADTs and RDFs, gives us a set of abstract CHCs modulo ADTs and UF.¹ The result of applying the 1-instantiation abstraction to the CHCs in Fig. 3.2c is shown in Fig. 3.3a. We show that, similar to the other transformations, the *k*-instantiation abstraction of CHCs is equi-satisfiable to the original CHCs, but, more importantly, it has more solutions than relationified CHCs. In fact, there is a hierarchy: all solutions to $(k - 1)$ -instantiation abstraction are also solutions to *k*-instantiation abstraction (Fig. 3.4), and all of them are solutions to the original CHCs. Furthermore, validating solutions of the abstraction of the CHCs is decidable, as it only involves ADTs and UF, but no RDFs. Hence, the *k*-instantiation abstraction is particularly useful for finding solutions. Note that while the abstraction makes validating solutions decidable, relationified predicates provide the supporting inductive lemmas (about RDFs) that are often necessary for solutions to be validated.

The uninterpreted function abstraction removes all occurrences of UF from a given formula (a detailed definition is given in Sec. 3.4). Thus, it results in even simpler CHCs — without UFs. An example of the uninterpreted functions abstraction of the CHCs in Fig. 3.3a is shown in Fig. 3.3b. The uninterpreted function abstraction also preserves (un)satisfiability, thanks to the relationified predicates. This abstraction is useful for proving unsatisfiability of the original CHCs, as refutational completeness of first order logic ensures the existence of a (finite) refutation proof for the abstraction.

While the description above might suggest that we explicitly construct the abstractions, this is not the case. The abstractions are tightly integrated as part of an IC3-style algorithm, RACER (Alg. 8 in Sec. 3.6). RACER uses *k*-instantiation abstraction to block infeasible counterexamples and to infer useful inductive lemmas over the predicates (including the predicates that define RDFs), and uninterpreted function abstraction to extend partial counterexamples (until they are either shown to be infeasible or become complete). The value for *k*, in both abstractions, is adjusted

¹Technically, the result is not in FOL because UFs are implicitly universally quantified. See Sec. 3.5 for a precise definition.

dynamically by RACER. While RACER does not guarantee to terminate — the underlying problem, after all, is undecidable, it does guarantee to find a counterexample if one exists.

For our running example, RACER produces the following solution:

$$Inv(y, i, j) \triangleq i \approx j \qquad Length(y, j) \triangleq j \geq 0$$

Note that the solution for *Length* is an inductive property of the *length* RDF.

3.2 CHCs modulo ADTs and Catamorphisms

In this section, we present the form of CHCs we are interested in. We also list out a number of assumptions so that the presentation in later parts of this section is easier to follow. We encode programs that manipulate datatypes as CHCs modulo a theory $\mathcal{T}$ that includes the theory of ADTs. The encoding further uses catamorphisms, specified via function axioms. In the sequel, we fix a signature Σ that consists of the sorts $\mathbf{s}$, $\mathbf{r}$, and $\text{List}_{\mathbf{s}}$, and constructors, selectors, and testers for lists. The signature Σ also includes a function symbol f of sort $\text{List}_{\mathbf{s}} \mapsto \mathbf{r}$ that is used to specify a catamorphism. We denote by $\varphi_{(\mathbf{b}, \oplus)}$ the catamorphism axiom for f .

We consider satisfiability of $C_f \wedge \varphi_{(\mathbf{b}, \oplus)}$, where C_f is a set of CHCs of the form:

$$\begin{aligned} & Init(\bar{y}, \bar{z}) \wedge \left(\bigwedge_i f(y_i) \approx z_i \right) \Rightarrow Inv(\bar{y}', \bar{z}') \\ & Inv(\bar{y}, \bar{z}) \wedge Tr(\bar{y}, \bar{z}, \bar{y}', \bar{z}') \wedge \left(\bigwedge_i f(y_i) \approx z_i \wedge f(y'_i) \approx z'_i \right) \Rightarrow Inv(\bar{y}', \bar{z}') \quad (3.1) \\ & Inv(\bar{y}, \bar{z}) \wedge \left(\bigwedge_i f(y_i) \approx z_i \right) \wedge Bad(\bar{y}, \bar{z}) \Rightarrow \perp \end{aligned}$$

where $y_i \in \bar{y}$, $z_i \in \bar{z}$, y_i is of sort $\text{List}_{\mathbf{s}}$ and z_i is of sort $\mathbf{r}$. Repetitions are allowed: y_i can be the same as y_j . We further assume that *Init*, *Tr*, and *Bad* do not contain any f applications, hence the bodies of the CHCs are f -pure. Such a form may be obtained by purification. $Inv \notin \Sigma$ is the only unknown predicate. We use C_f to refer to CHCs of the form Eq. (3.1), and we refer to $C_f \wedge \varphi_{(\mathbf{b}, \oplus)}$ as CHCs modulo function axiom.

CHCs like those in Eq. (3.1) can be directly constructed from functional programs and imperative programs.

The assumptions about the sort of y_i , z_i , f , that there is only one ADT, and one f are for presentation purposes only. We stress that our ideas and our implementation work for multiple catamorphisms defined over any ADT, mutually recursive ADTs,

mutually recursive catamorphisms, and non-linear CHCs. Relaxing these assumptions does not require any new algorithmic insights, and, in the implementation required no change from the underlying CHC solver.

Solutions We limit our attention to $\mathcal{A}$ -solutions of C_f modulo $\varphi_{(b,\oplus)}$, where $\mathcal{A}$ is the class of quantifier-free formulas over the signature $\Sigma \setminus \{f\}$. That is, f is not allowed to appear in the solution. We leave lifting this assumption to future work. Solutions that involve f can be accounted for, even with the restricted language, if the CHCs are constructed in such a way that the constraints in the clauses contain all the necessary f applications to prove safety (i.e., to obtain a solution), and that these applications are captured by variables that are included as arguments to solutions. In particular, term abstraction [2] may be used to introduce auxiliary variables that represent f -applications that are needed in order to express the solution, and augment the bodies of the CHCs with them. For example, in Fig. 3.2, the variable j that is added as an argument to Inv is an auxiliary variable introduced to express $length(y)$.

Motivation for our work Existing solvers for CHCs make multiple satisfiability queries over the background theory. Unfortunately, satisfiability becomes undecidable when considering the theory of ADTs with RDFs. The remainder of the chapter is dedicated to showing how existing solvers may be extended to handle such CHCs while ensuring that all satisfiability queries remain decidable.

3.3 Reducing CHCs modulo Catamorphisms to CHCs

To alleviate the problems arising from undecidability of catamorphisms, we show how to reduce CHCs modulo catamorphisms to pure CHCs whose underlying theory is decidable. Specifically, we show how to reduce $C_f \wedge \varphi_{(b,\oplus)}$, a set of CHCs with an axiomatized catamorphism f , to an equisatisfiable set of CHCs C_F without f . The key idea is to encode f by a fresh predicate (relation) symbol F , of appropriate sort, i.e., if f is of sort $List_s \mapsto r$, then F is a predicate over sorts $List_s \times r$. The predicate F is added to the set of unknown CHC predicates (which in our case consists of Inv only), and is used instead of f in C_f . The challenge is to express the axiom for f ($\varphi_{(b,\oplus)}$) as a CHC over F , while preserving equisatisfiability.

From CHCs with f to CHCs with F We first address the CHCs C_f . As the bodies of these CHCs are f -pure, we can use a simple substitution to replace f by F .

Definition 3 (Relationification) Let C_f be a set of CHCs whose bodies are f -pure. We denote by $C_f[f \mapsto F]$ the set of CHCs obtained by replacing all equalities of the form $f(t) \approx z$ in the constraints of C_f , with $F(t, z)$. $\square$

The CHCs resulting from relationification do not contain any occurrences of the function symbol f . On the other hand, they contain the new unknown predicate F . For example, relationification of the CHC $\forall x, y, z. A(x) \wedge f(x) \approx y \wedge f(y) \approx z \Rightarrow B(z)$ gives us $\forall x, y, z. A(x) \wedge F(x, y) \wedge F(y, z) \Rightarrow B(z)$.

The following lemma outlines the conditions in which relationification preserves satisfiability:

Lemma 1 *Let M be a model, and $\mathbf{f}$ and $\mathbf{F}$ be interpretations of f and F over the universe of M , respectively.*

- *If $M[f \mapsto \mathbf{f}] \models C_f$ and $\mathbf{F}$ is the graph² of $\mathbf{f}$ then $M[F \mapsto \mathbf{F}] \models C_f[f \mapsto F]$.*
- *If $M[F \mapsto \mathbf{F}] \models C_f[f \mapsto F]$ and $\mathbf{F}$ is an over approximation of the graph of $\mathbf{f}$ then $M[f \mapsto \mathbf{f}] \models C_f$.* $\square$

PROOF The first statement is straight forward. For the second statement, we have $M[F \mapsto \mathbf{F}] \models C_f[f \mapsto F]$. Let $\mathbf{G}_f$ be a stronger relation than $\mathbf{F}$. Since F occurs only on the tail of clauses in $C_f[f \mapsto F]$, we have that $M[F \mapsto \mathbf{G}_f] \models C_f[f \mapsto F]$. In particular, $\mathbf{G}_f$ can be the relation $\{(\mathbf{x}, \mathbf{z}) \mid \mathbf{f}(\mathbf{x}) = \mathbf{z}\}$, i.e., the graph of $\mathbf{f}$. This implies that $M[f \mapsto \mathbf{f}] \models C_f$. $\blacksquare$

From function axioms to CHCs Now, we move on to constructing CHCs to capture the catamorphism axiom.

Definition 4 (Positive relation constraint) Given a catamorphism axiom $\varphi_{(b, \oplus)}$ for f and a fresh relation symbol F , the positive relation constraint for $\varphi_{(b, \oplus)}$ is

$$RC_F^+(\varphi_{(b, \oplus)}) = \left(\begin{array}{c} \forall x, z. x \approx nil \wedge z \approx b \Rightarrow F(x, z) \wedge \\ \forall x, l, z. x \not\approx nil \wedge F(tail(x), l) \wedge z \approx l \oplus head(x) \Rightarrow F(x, z) \end{array} \right) \quad (3.2)$$

$\square$

The positive relation constraint corresponding to a catamorphism definition consists of one CHC for the base case and one CHC for the recursive case.

²The *graph* of a function $\mathbf{f}$ is the relation $\mathbf{F} \approx \{(x, \mathbf{f}(x)) \mid x \in dom(\mathbf{f})\}$, where $dom(\mathbf{f})$ is the domain of $\mathbf{f}$.

Example 6 Consider the axiom for the catamorphism $length$, $\varphi_{(0,+1)}$, from Ex. 1. Let $Length$ be a fresh relation symbol. The positive relation constraint for $\varphi_{(0,+1)}$ is:

$$\begin{aligned} & \forall x, z \cdot x \approx nil \wedge z \approx 0 \Rightarrow Length(x, z) \wedge \\ & \forall x, l, z \cdot x \not\approx nil \wedge Length(tail(x), l) \wedge z \approx l + 1 \Rightarrow Length(x, z) \end{aligned} \quad \square$$

Despite seemingly capturing only one direction of the definition encapsulated in the catamorphism axiom $\varphi_{(b,\oplus)}$, Theorem 1 shows that the positive relation constraint $RC_F^+(\varphi_{(b,\oplus)})$ does not allow spurious solutions when F is used instead of f in C_f , hence preserving satisfiability in both directions. This results from the following lemma that ensures that f is always a solution to $RC_F^+(\varphi_{(b,\oplus)})$ and that all solutions must overapproximate f .

Lemma 2 ($\mathbf{F}$ over approximates $\mathbf{f}$) *Let M be a model and $\mathbf{f}$ an interpretation of f over the universe of M . If $M[f \mapsto \mathbf{f}] \models \varphi_{(b,\oplus)}$ then*

1. *The relation $\mathbf{F}$ over the universe of M defined by the graph of $\mathbf{f}$ satisfies $M[F \mapsto \mathbf{F}] \models RC_F^+(\varphi_{(b,\oplus)})$.*
2. *For every relation $\mathbf{F}$ over the universe of M , if $M[F \mapsto \mathbf{F}] \models RC_F^+(\varphi_{(b,\oplus)})$ then $\mathbf{F}$ is an over approximation of the graph of $\mathbf{f}$.* $\square$

PROOF Part 1 is straightforward. As for part 2, let $List_s$ denote the set of all elements of sort $List_s$ in the universe of M . Wlog, we can assume that $List_s$ consists of all constructor terms (see Sec. 1.2.3). Therefore, we can prove 2 by straight forward induction on the elements of $List_s$: For the base case, since $\mathbf{f}(nil) = \mathbf{b}$, we have to show that $(nil, \mathbf{b}) \in \mathbf{F}$. We can see that the clause $\forall x, z \cdot x \approx nil \wedge b \approx z \Rightarrow F(x, z)$ enforces this. For the recursive case, assume that $\mathbf{f}(\mathbf{y}) = \mathbf{l}$ and $(\mathbf{y}, \mathbf{l}) \in \mathbf{F}$. Let $\mathbf{x} = insert(\mathbf{k}, \mathbf{y})$ and $\mathbf{f}(\mathbf{x}) = \mathbf{z}$. It has to be the case that $\mathbf{z} = \mathbf{k} \oplus \mathbf{l}$. The clause $\forall x, l, z \cdot x \not\approx nil \wedge F(tail(x), l) \wedge z \approx head(x) \oplus l \Rightarrow F(x, z)$ ensures that $(\mathbf{x}, \mathbf{z}) \in \mathbf{F}$. Thus, all tuples $(\mathbf{x}, \mathbf{z})$ such that $\mathbf{f}(\mathbf{x}) = \mathbf{z}$ must belong to the relation $\mathbf{F}$. $\blacksquare$

We are now ready to define the CHCs obtained after eliminating f :

Definition 5 (Relationified CHCs with CHC encoding of catamorphisms) Given C_f and $\varphi_{(b,\oplus)}$, the set of CHCs C_F obtained by relationification of C_f and reducing the function axiom $\varphi_{(b,\oplus)}$ to CHCs is

$$C_F = RC_F^+(\varphi_{(b,\oplus)}) \wedge C_f[f \mapsto F] \quad \square$$

Theorem 1 *The set of CHCs after the transformation is equisatisfiable to the original CHCs modulo catamorphisms: C_F is equisatisfiable with $\varphi_{\langle b, \oplus \rangle} \wedge C_f$.* $\square$

PROOF Left to right direction. Let M be a model for $RC_F^+(\varphi_{\langle b, \oplus \rangle}) \wedge C_f[f \mapsto F]$, and denote $M[F]$ by $\mathbf{F}$. Let $\mathbf{f}$ be a function over the universe of M such that $M[f \mapsto \mathbf{f}] \models \varphi_{\langle b, \oplus \rangle}$. $\mathbf{f}$ exists because $\varphi_{\langle b, \oplus \rangle}$ defines a catamorphism. We know from Lem. 2 that $\mathbf{F}$ over approximates the graph of $\mathbf{f}$. Therefore, by Lem. 1 we can see that $M[f \mapsto \mathbf{f}]$ is a model for $\varphi_{\langle b, \oplus \rangle} \wedge C_f$ as well.

Right to Left direction. Let M be a model for $\varphi_{\langle b, \oplus \rangle} \wedge C_f$. Let $M[f] = \mathbf{f}$ and let $\mathbf{F}$ be the graph of $\mathbf{f}$. By Lem. 2, we know that $M[F \mapsto \mathbf{F}] \models RC_F^+(\varphi_{\langle b, \oplus \rangle})$. Therefore, by Lem. 1, $M[F \mapsto \mathbf{F}] \models C_f[f \mapsto F]$. $\blacksquare$

Applying the transformation on CHCs C_f of the form given in Eq. (3.1), we get:

$$\begin{aligned}
& y \approx \text{nil} \wedge z \approx b \Rightarrow F(y, z) \\
& y \not\approx \text{nil} \wedge F(\text{tail}(y), l) \wedge y \approx l \oplus \text{head}(x) \Rightarrow F(y, z) \\
& \text{Init}(\bar{y}, \bar{z}) \wedge (\bigwedge_i F(y_i, z_i)) \Rightarrow \text{Inv}(\bar{y}', \bar{z}') \\
& \text{Inv}(\bar{y}, \bar{z}) \wedge \text{Tr}(\bar{y}, \bar{z}, \bar{y}', \bar{z}') \wedge (\bigwedge_i F(y_i, z_i) \wedge F(y'_i, z'_i)) \Rightarrow \text{Inv}(\bar{y}', \bar{z}') \\
& \text{Inv}(\bar{y}, \bar{z}) \wedge (\bigwedge_i F(y_i, z_i)) \wedge \text{Bad}(\bar{y}, \bar{z}) \Rightarrow \perp
\end{aligned} \tag{3.3}$$

While the transformation preserves satisfiability, the following example demonstrates that it may not preserve $\mathcal{A}$ -solutions (Def. 2):

Example 7 (The transformation does not preserve solutions) The set of CHCs in Eq. (3.4) encode the safety of a program that removes all the elements from a list y while keeping track of its length i . We use term abstraction to encode $\text{length}(y)$ via an auxiliary integer variable j , such that neither y nor $\text{length}(y)$ are needed in the inductive invariant. Accordingly, the inductive invariant is specified over i, j (note that y is also omitted from Inv in this example). The first clause initializes variable i to the length of y and the second clause keeps track of the length of y as elements are removed from the list. The third clause derives false if i is non zero once all elements are removed.

$$\begin{aligned}
& \text{length}(y) \approx i \Rightarrow \text{Inv}(i, i) \\
& \text{Inv}(i, j) \wedge \text{length}(y) \approx j \wedge y \not\approx \text{nil} \wedge y' \approx \text{tail}(y) \wedge \text{length}(y') \approx j' \Rightarrow \text{Inv}(i-1, j') \\
& \text{Inv}(i, j) \wedge \text{length}(y) \approx j \wedge y \approx \text{nil} \wedge i \not\approx 0 \Rightarrow \perp
\end{aligned} \tag{3.4}$$

The set of CHCs in Eq. (3.4) is satisfiable modulo $\varphi_{\langle 0, +1 \rangle}$, the catamorphism axiom for length , with the solution: $\text{Inv}(i, j) = i \approx j$, defined over a signature that excludes

length. The equi-satisfiable CHC $C_F = RC_F^+(\varphi_{\langle 0, +1 \rangle}) \wedge C[\textit{length} \mapsto \textit{Length}]$ is

$$\begin{aligned}
& y \approx \textit{nil} \wedge z \approx 0 \Rightarrow \textit{Length}(y, z) \\
& y \not\approx \textit{nil} \wedge \textit{Length}(\textit{tail}(y), l) \wedge z \approx 1 + l \Rightarrow \textit{Length}(y, z) \\
& \textit{Length}(y, i) \Rightarrow \textit{Inv}(i, i) \\
& \textit{Inv}(i, j) \wedge \textit{Length}(y, j) \wedge y \not\approx \textit{nil} \wedge y' \approx \textit{tail}(y) \wedge \textit{Length}(y', j') \Rightarrow \textit{Inv}(i - 1, j') \\
& \textit{Inv}(i, j) \wedge \textit{Length}(y, j) \wedge y \approx \textit{nil} \wedge i \not\approx 0 \Rightarrow \perp
\end{aligned} \tag{3.5}$$

However, it does not have a solution in which the interpretation for *Length* is expressible without *length*. Intuitively, this is because the interpretation for *Length* in a solution needs to be exactly the graph of the interpretation of *length*, while the relation corresponding to *length* cannot be expressed in first order logic. We formalize this intuition next. Assume to the contrary that there is a model M for the transformed CHCs in which the interpretation of *Length*, **Length**, is not equal to the graph of *length*, where *length* is the (unique) interpretation of *length* over the universe of M . Then, **Length** has to be an over approximation of the graph of *length* (due to Lem. 2). Take a pair (y, j) that is in **Length** but not in the graph of *length*, i.e., $\textit{length}(y) \neq j$, such that y is minimal. We show that the last CHC must be violated. If $y = \textit{nil}$, use the third clause to deduce that $(j, j) \in \textit{Inv}$, which immediately implies that the last clause is violated (since by our assumption $j \neq 0$ in this case). Otherwise, consider $y' = \textit{tail}(y)$ and $j' = \textit{length}(y')$. Since **Length** overapproximates *length*, $(y', j') \in \textit{Length}$ as well. From the 4th clause, we deduce that $(j - 1, j') \in \textit{Inv}$, even though $j - 1 \neq j'$ (recall that $j \neq \textit{length}(y)$ whereas $\textit{length}(y) = \textit{length}(y') + 1 = j' + 1$). From $(j - 1, j') \in \textit{Inv}$, using the pairs in the graph of *length*, we can use the 4th clause to iteratively decrease both elements of the pair by 1, ultimately showing that there exists $i \neq 0$ such that $(i, 0) \in \textit{Inv}$. This again contradicts the last clause (by taking the pair $(\textit{nil}, 0) \in \textit{Length}$). $\square$

Remark 1 We have shown how to relationify catamorphisms on lists. Extending this to catamorphisms on other ADTs is straightforward: relationification of f introduces a Horn clause over the uninterpreted predicate F for each of the base and recursive cases.

The approach also works for RDFs over ADTs that are defined using SMT-LIB syntax. In this case, an RDF is defined by a single axiom: $\forall \bar{v} \cdot f(\bar{v}) \approx t(\bar{v})$. To perform relationification, we further require that the base and recursive cases can be easily identified. This is already used by the majority of SMT-solvers supporting RDFs. For soundness, we require that the axiom is satisfiable (i.e., actually defines a function). This requirement is omitted from SMT-LIB, but is common in many other systems that allow user-defined recursive functions (e.g., Dafny [75]). In general, our approach applies to any RDF that supports an appropriate relationification procedure satisfying Lem. 2. $\square$

To conclude, in this section, we presented a transformation that takes as input a set of CHCs modulo RDFs, $C_f \wedge \varphi_{\langle b, \oplus \rangle}$, and produces a set of CHCs without RDFs, C_F . The transformation preserves satisfiability (theorem 1) but not solutions (Ex. 7). Note that all solutions to C_F are solutions to $C_f \wedge RC_F^+(\varphi_{\langle b, \oplus \rangle})$.

3.4 Introducing RDF Abstractions into CHCs

In this section we tackle the potential loss of CHC solutions without compromising decidability of the background theory by using abstractions of the RDFs.

We start by re-introducing the RDF with the axiom that defines it, thus recovering solutions. Formally, instead of substituting $f(x) \approx z$ by $F(x, z)$ during relationification, we substitute it by $f(x) \approx z \wedge F(x, z)$. Let $C_f[f \mapsto (f \wedge F)]$ denote this transformation.

Definition 6 (Relationified CHCs modulo RDFs) Given C_f and $\varphi_{\langle b, \oplus \rangle}$, the set of CHCs C_{fF} obtained after relationification, reducing function axiom $\varphi_{\langle b, \oplus \rangle}$ to CHCs and re-introducing RDFs is

$$C_{fF} = RC_F^+(\varphi_{\langle b, \oplus \rangle}) \wedge C_f[f \mapsto (f \wedge F)] \quad \square$$

Specifically, when C_f is Eq. (3.3) modulo $\varphi_{\langle b, \oplus \rangle}$, C_{fF} is,

$$\begin{aligned} & \psi_b(y, z) \Rightarrow F(y, z) \\ & \psi_r(y, z)[f \mapsto F] \Rightarrow F(y, z) \\ & Init(\bar{y}, \bar{z}) \wedge (\bigwedge_i F(y_i, z_i) \wedge f(y_i) \approx z_i) \Rightarrow Inv(\bar{y}', \bar{z}') \\ & Inv(\bar{y}, \bar{z}) \wedge Tr(\bar{y}, \bar{z}, \bar{y}', \bar{z}') \wedge (\bigwedge_i F(y_i, z_i) \wedge f(y_i) \approx z_i \wedge F(y'_i, z'_i) \wedge f(y'_i) \approx z'_i) \Rightarrow Inv(\bar{y}', \bar{z}') \\ & Inv(\bar{y}, \bar{z}) \wedge (\bigwedge_i F(y_i, z_i) \wedge f(y_i) \approx z_i) \wedge Bad(\bar{y}, \bar{z}) \Rightarrow \perp \end{aligned} \quad (3.6)$$

This transformation preserves not only satisfiability, but also solutions, in the following sense:

Theorem 2 *If $M \models \varphi_{\langle b, \oplus \rangle} \wedge C_f$ then $M[F \mapsto \top] \models \varphi_{\langle b, \oplus \rangle} \wedge C_{fF}$. If $M \models \varphi_{\langle b, \oplus \rangle} \wedge C_{fF}$, then $M \models \varphi_{\langle b, \oplus \rangle} \wedge C_f$.* $\square$

PROOF First part is straightforward; second part follows from $M[f] \subseteq M[F]$ (Lem. 2). ■

Corollary 1 $\varphi_{\langle b, \oplus \rangle} \wedge C_f$ is equisatisfiable with $\varphi_{\langle b, \oplus \rangle} \wedge C_{fF}$. $\square$

From theorem 2 we can see that, if there is an $\mathcal{A}$ -solution to $\varphi_{\langle b, \oplus \rangle} \wedge C_f$ then there will also be an $\mathcal{A}$ -solution to $\varphi_{\langle b, \oplus \rangle} \wedge C_{fF}$ (assuming that $\top \in \mathcal{A}$).

Unfortunately, re-introducing the catamorphisms with their axioms also makes the underlying logic undecidable. To mitigate this, we work over abstractions of the catamorphism. In the remainder of the section, we define the abstractions we use (Sec. 3.4) and explain what impact the abstractions have on the solutions to the CHCs (Sec. 3.5). In Sec. 3.6, we describe an algorithm for checking the satisfiability of CHCs modulo RDFs using these abstractions.

Abstraction of RDF

In this section, we define two abstractions of formulas modulo RDFs. The first abstraction unfolds RDF applications using the axioms of RDF and then replaces any remaining RDF applications with uninterpreted functions and the second abstraction further replaces the literals containing uninterpreted functions with $\top$. Both of the abstractions result in weaker formulas, and can be understood as over approximating the formulas to which they are applied. As we will see later, these abstractions allow us to stay in a decidable fragment when validating solutions, while guaranteeing that we have potentially more solutions than C_F .

To define the abstractions precisely, we first define *prenex f -pure* formulas. A formula is *prenex f -pure* if it is of the form

$$\exists \bar{z} \cdot \left(\bigwedge_i f(e_i) \approx z_i \right) \wedge \theta'(\bar{x}, \bar{z}) \quad (3.7)$$

where each e_i is a term over free variables $\bar{x}$, variables $\bar{z}$, and containing no f , each z_i is a variable in $\bar{z}$, and θ' is a quantifier free formula over $\bar{x}$ and $\bar{z}$ containing no f terms. Every quantifier-free formula θ can be converted into an equivalent prenex f -pure formula by the purification procedure described in Sec. 1.1: introducing existentially quantified variables that represent the f applications. For example, $p(f(f(x)))$ may be rewritten into $\exists y, z \cdot f(x) \approx y \wedge f(y) \approx z \wedge p(z)$.

Instantiating function axiom Given an RDF f together with its function axiom $\varphi = \forall x \cdot f(x) \approx t(x)$, and a prenex f -pure formula θ , *instantiation* of f in θ is the formula obtained by conjoining the literal $t[x \mapsto e_i] \approx z_i$ with each literal $f(e_i) \approx z_i$ in θ . The k -*instantiation* of f on θ is the formula defined recursively as follows: the 0-instantiation is θ itself and $(i + 1)$ -instantiation is the instantiation of f on the i -instantiation (after the i -instantiation is brought to prenex f -pure form).

Definition 7 (k -instantiation abstraction) Given an RDF f and a prenex f -pure formula θ , the k -instantiation abstraction of f on θ , denoted by $\alpha_{rf}^k(\theta)$, is the formula

obtained by replacing all f occurrences in the k -instantiation of θ with a fresh uninterpreted function. $\square$

Example 8 Let $\theta = \text{length}(x) \approx z_1 \wedge \text{length}(y) \approx z_2 \wedge y \approx \text{tail}(x) \wedge z_1 \approx 2 \wedge z_2 \not\approx 1$. Then,

$$\begin{aligned}\alpha_{rf}^0(\theta) &= \text{length}_{uf}(x) \approx z_1 \wedge \text{length}_{uf}(y) \approx z_2 \wedge y \approx \text{tail}(x) \wedge z_1 \approx 2 \wedge z_2 \not\approx 1 \\ \alpha_{rf}^1(\theta) &= \alpha_{rf}^0(\theta) \wedge \text{ite}(x \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(x))) \approx z_1 \wedge \\ &\quad \text{ite}(y \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(y))) \approx z_2 \\ \alpha_{rf}^2(\theta) &= \exists l_{tx}, l_{ty} \cdot \text{length}_{uf}(\text{tail}(x)) \approx l_{tx} \wedge \text{length}_{uf}(\text{tail}(y)) \approx l_{ty} \wedge \quad \square \\ &\quad \alpha_{rf}^0(\theta) \wedge \text{ite}(x \approx \text{nil}, 0, 1 + l_{tx}) \approx z_1 \wedge \text{ite}(y \approx \text{nil}, 0, 1 + l_{ty}) \approx z_2 \wedge \\ &\quad \text{ite}(\text{tail}(x) \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(\text{tail}(x)))) \approx l_{tx} \wedge \\ &\quad \text{ite}(\text{tail}(y) \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(\text{tail}(y)))) \approx l_{ty}\end{aligned}$$

Definition 8 (Uninterpreted function abstraction) Given an uninterpreted function symbol f , and a prenex f -pure formula θ , the uninterpreted function abstraction, denoted $\alpha^\uparrow(\theta)$, is obtained by replacing all literals containing f in θ with $\top$. $\square$

We write $\alpha_{rf}^{k\uparrow}(\theta)$ for $(\alpha_{rf}^\uparrow \circ \alpha_{rf}^k)(\theta)$.

Example 9 The result of applying the uninterpreted function abstraction for the k -unrolling abstractions in Ex. 8 is

$$\begin{aligned}\alpha_{rf}^{0\uparrow}(\theta) &= y \approx \text{tail}(x) \wedge z_1 \approx 2 \wedge z_2 \not\approx 1 \\ \alpha_{rf}^{1\uparrow}(\theta) &= \exists l_{tx}, l_{ty} \cdot \alpha_{rf}^{0\uparrow}(\theta) \wedge \text{ite}(x \approx \text{nil}, 0, 1 + l_{tx}) \approx z_1 \wedge \text{ite}(y \approx \text{nil}, 0, 1 + l_{ty}) \approx z_2 \\ \alpha_{rf}^{2\uparrow}(\theta) &= \exists l_{tx}, l_{ty}, l_{ttx}, l_{tty} \cdot \alpha_{rf}^{0\uparrow}(\theta) \wedge \text{ite}(x \approx \text{nil}, 0, 1 + l_{tx}) \approx z_1 \wedge \text{ite}(y \approx \text{nil}, 0, 1 + l_{ty}) \approx z_2 \wedge \\ &\quad \text{ite}(\text{tail}(x) \approx \text{nil}, 0, 1 + l_{ttx}) \approx l_{tx} \wedge \text{ite}(\text{tail}(y) \approx \text{nil}, 0, 1 + l_{tty}) \approx l_{ty}\end{aligned}$$

Notice that $\alpha_{rf}^{1\uparrow}(\theta)$ has additional, existentially quantified, variables not present in $\alpha_{rf}^1(\theta)$. These variables are introduced during purification. Similarly, the additional variables in $\alpha_{rf}^{2\uparrow}(\theta)$ have also been produced during purification. $\square$

The abstractions provide overapproximations of θ in the sense that they may only increase the set of satisfying models:

Lemma 3 Let f be an RDF, θ a prenex f -pure formula and M a model. Then, for any k ,

1. $M \models \theta \Rightarrow M[f_{uf} \mapsto M[f]] \models \alpha_{rf}^k(\theta)$, and
2. $M \models \alpha_{rf}^k(\theta) \Rightarrow M \models \alpha_{rf}^{k\uparrow}(\theta)$. $\square$

3.5 CHCs modulo RDF vs CHCs over RDF abstractions

Recall that, originally we started with a set of CHCs C_f that contain applications of RDF f . We showed that replacing f with a fresh relation F preserves satisfiability (theorem 1) but not solutions (Ex. 7). Re-introducing f regains solutions (theorem 2), but also brings back the problem of undecidability. In this section, we analyze the effects of re-introducing *abstractions* of the RDF f into C_F . Recall that $C_F = RC_F^+(\varphi_{\langle b, \oplus \rangle}) \wedge C_f[f \mapsto F]$ is the transformed CHCs without f applications, and $C_{fF} = RC_F^+(\varphi_{\langle b, \oplus \rangle}) \wedge C_f[f \mapsto (f \wedge F)]$ is the CHCs we get after conjoining back f .

Using the k -instantiation abstraction When trying to establish that C_{fF} is satisfiable modulo $\varphi_{\langle b, \oplus \rangle}$, our algorithm uses the k -instantiation abstraction. Let C_{fF}^k be the result of replacing all constraints in the clauses of C_{fF} with their k -instantiation abstractions. (The resulting formulas may be normalized so that all the newly introduced existential quantifiers in the constraints are converted to universal quantifiers over the whole clause.) C_{fF}^k has uninterpreted functions. This means that satisfiability of C_{fF}^k allows the uninterpreted functions to receive *some* interpretation, while we are interested in solutions that are valid for *any* interpretation of the functions, including the ones that satisfy the abstracted function axioms (akin to an implicit second order universal quantifier over f). Therefore, C_{fF}^k is outside of the usual domain of CHCs, and we refrain from considering its satisfiability. Instead, we extend the notion of an $\mathcal{A}$ -solution (Def. 2) to the abstracted formula C_{fF}^k , and compare $\mathcal{A}$ -solutions of C_{fF}^k with $\mathcal{A}$ -solutions of the original set of CHCs. Recall that once the solution is plugged in, the requirement is that the resulting sentences are *valid*, which has the effect of implicitly universally quantifying over the uninterpreted function f . The next theorems show that the k -instantiation abstractions induce a hierarchy in terms of their sets of solution.

Theorem 3 *Let S be an $\mathcal{A}$ -solution to C_{fF}^k . Then, S is an $\mathcal{A}$ -solution to C_{fF} modulo $\varphi_{\langle b, \oplus \rangle}$.* $\square$

PROOF Let kC be the set of CHCs obtained by doing k -instantiation (no abstraction) of C_{fF} . By the properties of k -instantiation, we know that $kC \equiv C_{fF}$ modulo $\varphi_{\langle b, \oplus \rangle}$. Let ψ be the conjunction of the sentences we get from kC by replacing all uninterpreted predicates with their corresponding formulas from S . It suffices to show that ψ is valid modulo $\varphi_{\langle b, \oplus \rangle}$. Similarly, let ψ^k be the conjunction of sentences we get from C_{fF}^k by replacing all uninterpreted predicates with their corresponding formulas from S . ψ and ψ^k differ only in f being uninterpreted in ψ^k . Since ψ^k is valid, ψ has to be valid modulo $\varphi_{\langle b, \oplus \rangle}$. Thus, S is an $\mathcal{A}$ -solution to C_{fF} modulo $\varphi_{\langle b, \oplus \rangle}$. $\blacksquare$

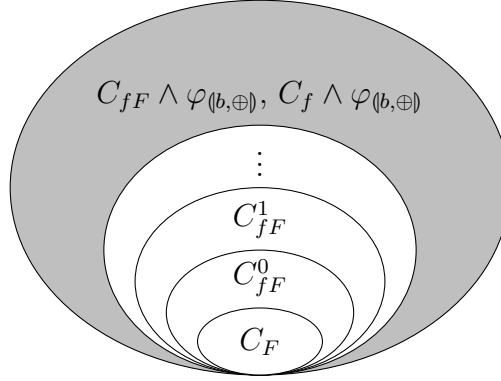


Figure 3.4: Comparing $\mathcal{A}$ -solutions of different abstractions of CHCs. Each ellipse represents the set of $\mathcal{A}$ -solutions to the corresponding formula. Validating solutions of CHC in the shaded region is undecidable.

Importantly, validating $\mathcal{A}$ -solutions to C_{fF}^k amounts to checking validity of universally quantified formulas (equivalently, checking unsatisfiability of quantifier-free formulas) modulo the background theory combined with uninterpreted functions, which is decidable.

Another interesting aspect of C_{fF}^k is that $\mathcal{A}$ -solutions are preserved as k increases:

Theorem 4 *If S is an $\mathcal{A}$ -solution to C_{fF}^k , then S is an $\mathcal{A}$ -solution to $C_{fF}^{k'}$, where $k' \geq k$.* $\square$

PROOF The constraints in the clauses of $C_{fF}^{k'}$ are stronger than the constraints in the corresponding clauses of C_{fF}^k . $\blacksquare$

Similarly, if S is an $\mathcal{A}$ -solution to C_F , then S is an $\mathcal{A}$ -solution to C_{fF}^k for any k . The hierarchy induced by the different abstractions is outlined in Fig. 3.4. The hierarchy implies that when searching for a solution to C_{fF} modulo $\varphi_{(b, \oplus)}$, we can start with C_{fF}^0 and unroll f more and more, potentially increasing the set of solutions. If we find a solution for any value of k , we have a solution to our original problem.

Example 10 Recall the set of CHCs from Ex. 7. We showed that the relationified

CHC C_F did not have an $\mathcal{A}$ -solution. However, the set of abstracted CHCs C_{fF}^1 :

$$\begin{aligned}
& y \approx \text{nil} \wedge z \approx 0 \Rightarrow \text{Length}(y, z) \\
& y \not\approx \text{nil} \wedge \text{Length}(\text{tail}(y), l) \wedge z \approx 1 + l \Rightarrow \text{Length}(y, z) \\
& \text{Length}(y, i) \wedge i \approx \text{length}_{uf}(y) \wedge i \approx \text{ite}(y \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(y))) \Rightarrow \text{Inv}(y, i, i) \\
& \text{Length}(y, j) \wedge j \approx \text{length}_{uf}(y) \wedge j \approx \text{ite}(y \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(y))) \wedge \\
& \text{Length}(y', j') \wedge j' \approx \text{length}_{uf}(y') \wedge j' \approx \text{ite}(y' \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(y'))) \wedge \\
& \text{Inv}(y, i, j) \wedge y \not\approx \text{nil} \wedge y' \approx \text{tail}(y) \Rightarrow \text{Inv}(y', i - 1, j') \\
& \text{Length}(y, j) \wedge j \approx \text{length}_{uf}(y) \wedge j \approx \text{ite}(y \approx \text{nil}, 0, 1 + \text{length}_{uf}(\text{tail}(y))) \wedge \\
& \text{Inv}(y, i, j) \wedge y \approx \text{nil} \wedge i \not\approx 0 \Rightarrow \perp
\end{aligned}$$

has a solution which maps Inv to the formula $i \approx j$ (as in the solution for C_f) and Length to the formula $\top$. Furthermore, validating this solution is decidable as the validation query is a quantifier free formula over a combination of ADTs, LIA, and UF. $\square$

Using the uninterpreted function abstraction When trying to establish that C_{fF} is unsatisfiable modulo $\varphi_{(b, \oplus)}$, our algorithm avoids the presence of uninterpreted functions in the CHCs by using the uninterpreted function abstraction.

Let $C_{fF}^{k\uparrow}$ be the result of replacing all constraints in the clauses of C_{fF} with their $\alpha_{rf}^{k\uparrow}$ abstractions. This abstraction preserves counterexamples:

Theorem 5 *If $C_{fF}^{k\uparrow}$ is unsatisfiable, then C_{fF} is unsatisfiable modulo $\varphi_{(b, \oplus)}$.* $\square$

PROOF We know from theorem 1 and theorem 2 that, modulo $\varphi_{(b, \oplus)}$, C_F is equisatisfiable to C_{fF} . We prove that if $C_{fF}^{k\uparrow}$ is unsatisfiable, then C_F is unsatisfiable. The contrapositive of this statement is that if C_F is satisfiable, then $C_{fF}^{k\uparrow}$ is satisfiable. The constraints in the clauses of $C_{fF}^{k\uparrow}$ are stronger than the constraints in the corresponding clauses of C_F . Therefore, for all models M , if $M \models C_F$, then $M \models C_{fF}^{k\uparrow}$. $\blacksquare$

3.6 Solving CHC Modulo ADT and RDF

In this section, we present RACER, an algorithm that solves CHCs of the form C_{fF} (Eq. (3.6)) modulo a catamorphism axiom $\varphi_{(b, \oplus)}$ for f . RACER is sound: if it finds a solution, it is a solution for the original set of CHCs, C_f , modulo the axiom; if it declares UNSAT, then the original set of CHCs modulo the axiom is UNSAT. Further, if C_f is UNSAT, RACER is guaranteed to terminate. RACER works by

Algorithm 8: The RACER algorithm. The parts of the algorithm which are highlighted in green (resp. yellow) are the only places that use α_{rf}^k (resp. $\alpha_{rf}^{k\uparrow}$) abstraction. Pink highlights the increment of k . The rest of the algorithm is the same as SPACER.

Input: A CHC of the form Eq. (3.6) Input: A catamorphism axiom $\varphi_{\langle b, \oplus \rangle}$ Output: SAT/UNSAT <pre> 1 $R^F := \{\langle nil, b \rangle\};$ 2 $\bar{O}_0^F = R^F; \bar{O}_i^F := \top, \forall i > 0$ 3 $R^I := \perp; \bar{O}_i^I := \top, \forall i \geq 0;$ 4 $N := 0; k := 0$ 5 create_pred_bad() 6 while $\top$ do 7 $\langle \theta, i+1, \mathcal{P} \rangle := Q.pop()$ 8 $\mathcal{I}, M, res := check_reach(\langle \theta, i, \mathcal{P} \rangle)$ 9 if res then 10 create_pred($\mathcal{I}, \langle \theta, i+1, \mathcal{P} \rangle, M$) 11 if is_bad_rch() then return UNSAFE 12 else 13 block_pob($\langle \theta, i+1, \mathcal{P} \rangle$) 14 $Q := Q \cup \langle \theta, i+2, \mathcal{P} \rangle$ 15 if is_bad_blk() then 16 k := k + 1 17 $N := N + 1$ 18 create_pred_bad() 19 if chk_ind() then return SAFE 20 function chk_ind(): 21 return $\exists j \cdot 0 \leq j \leq N-1 \wedge \bar{O}_{j+1}^I \Rightarrow \bar{O}_j^I \wedge \bar{O}_{j+1}^F \Rightarrow \bar{O}_j^F$ 22 function check_reach($\langle \theta, i, \mathcal{P} \rangle$): 23 $\beta := \alpha_{rf}^k(\mathcal{F}_{\mathcal{P}}(\bar{O}_i)) \wedge \varphi'$ 24 if $\neg \text{ISAT}(\beta)$ then return $\neg, \neg, \perp$ 25 $\mu := \{R^Q[\bar{v}] \mid \langle Q, \bar{v} \rangle \in \overline{UT}^{\mathcal{P}}\}$ 26 $M := \max_sat(\beta, \mu)$ 27 $\mathcal{I} := \text{Implicant}(\beta, M)$ 28 $\mathcal{I} := \mathcal{I} \wedge \{r \mid r \in \mu \wedge M \models r\}$ 29 return $\mathcal{I}, M, \top$</pre>	<pre> 30 function is_bad_rch(): 31 $\theta := R^I \wedge (\bigwedge_i R^F[y_i, z_i]) \wedge Bad$ 32 return ISAT(θ) 33 function block_pob($\langle \theta, i+1, \mathcal{P} \rangle$): 34 $\beta := \alpha_{rf}^k(\mathcal{F}_{\mathcal{P}}(\bar{O}_i))$ 35 $c := \text{ITP}(\beta, \theta')[\bar{x}' \mapsto \bar{x}]$ 36 $\ell := \text{IND_GEN}(c)$ 37 $\bar{O}_j^{\mathcal{P}} := \bar{O}_j^{\mathcal{P}} \wedge \ell \quad \forall j \leq i+1$ 38 function create_pred_bad(): 39 $\theta := \bar{O}_{N+1}^I \wedge (\bigwedge_i \bar{O}_{N+1}^F[y_i, z_i] \wedge \alpha_{rf}^{k\uparrow}(f(y_i) \approx z_i)) \wedge Bad$ 40 if $\neg \text{ISAT}(\theta)$ then return 41 Let $M \models \theta; \bar{v} := \text{vars}(\theta)$ 42 for $\langle Q, \bar{y} \rangle \in \overline{UT}^{Bad}$ do 43 pob := MBP($\bar{v} \setminus \{\bar{y}\}, \theta, M$) $Q := Q \cup \langle pob, N+1, Q \rangle$ 44 function create_pred($\mathcal{I}, \langle \theta, i+1, \mathcal{P} \rangle, M$): 45 $\Phi := \{\langle Q, \bar{u} \rangle \mid \langle Q, \bar{u} \rangle \in \overline{UT}^{\mathcal{P}} \wedge M \not\models R^Q[\bar{u}]\}$ 46 if $\Phi = \emptyset$ then 47 // compute successor 48 $\bar{v} := \text{vars}(\mathcal{I}) \setminus \{\bar{x}'\}$ 49 $r := \text{MBP}(\bar{v}, \alpha_{rf}^{k\uparrow}(\mathcal{I} \wedge \theta'), M)$ 50 $R^{\mathcal{P}} := R^{\mathcal{P}} \vee r[\bar{x}' \mapsto \bar{x}]$ 51 else 52 kids := $\emptyset$ 53 for $\langle Q, \bar{u} \rangle \in \Phi$ do 54 $\gamma := \text{MBP}(\bar{v}, \alpha_{rf}^{k\uparrow}(\mathcal{I} \wedge \theta'), M)$ 55 $\gamma := \gamma[\bar{u} \mapsto \bar{x}]$ 56 kids := kids $\cup \langle \gamma, i, Q \rangle$ 57 $Q := Q \cup kids$ 58 $Q := Q \cup \langle \theta, (i+1), \mathcal{P} \rangle$ 59 function is_bad_blk(): 60 $\theta := \bar{O}_N^I \wedge (\bigwedge_i \bar{O}_N^F[y_i, z_i] \wedge \alpha_{rf}^k(f(y_i) \approx z_i)) \wedge Bad$ 61 return $\neg \text{ISAT}(\theta)$</pre>
--	---

simultaneously searching for $\mathcal{A}$ -solutions of C_{fF}^k as well as counter examples to $C_{fF}^{k\uparrow}$ while periodically increasing k . RACER is based on SPACER (see Sec. 1.8). The input to RACER (Alg. 8) are CHCs of the form Eq. (3.6) together with a function axiom for f and the output is SAFE/UNSAFE. RACER need not terminate. CHCs like those in Eq. (3.6) have exactly 2 uninterpreted predicates: I , the uninterpreted predicate representing the desired inductive invariant of the program or F , the relationified RDF. Alg. 8 is an instantiation of Alg. 3 for just 2 predicates augmented with RACER specific steps (highlighted).

The Initialization phase RACER initializes both R^F and $\bar{O}_0^F$ to $\{(nil, b)\}$ (Line 1). Note that we are using a set of states to represent a formula over variables $\bar{x}$. Next, R^I is initialized with $\perp$ and all the $\bar{O}^I$ with $\top$. These initialization steps are sound because in Eq. (3.6), F is the only predicate that appears in the head of a clause that has no predicate in the body. After initialization, RACER creates predecessors to Bad and add them to the queue (line 5).

IC3-part of the algorithm Just like SPACER (Alg. 3, Sec. 1.8), RACER iteratively calls the *check_reach* function to check the reachability of a POB at a particular level (line 8). The *check_reach* function first checks whether the POB is blocked using may summaries (line 24). If the check is satisfiable, the function finds a model that satisfies as many must summaries as possible using a *max_sat* algorithm. The *check_reach* function returns $\top$ only if the may summaries are not strong enough to block the POB at this level. In this case, the algorithm calls the *create_pred* function to create and add predecessor POBs to the queue (line 10). Otherwise, it calls the *block_pob* to learn a lemma that blocks the POB (line 13). The *create_pred* function (line 44) uses Model Based Projection (MBP, see Sec. 1.3) to create a predecessor POB for all the predicates in the tail whose must summaries are *not* satisfied by the model (but their may summaries are). If there are no such predicates, the POB is reachable from the must summaries. In this case, the algorithm computes a reachable state that intersects with the POB using MBP (line 48) and adds it to the set of reachable states. RACER routinely checks whether Bad is reachable (line 11) and increases the depth of exploration when Bad is blocked (line 15).

The role of abstractions When checking for one-step reachability, RACER uses the k -instantiation abstraction to abstract the predicate transformer (line 23). RACER employs the same abstraction when learning a lemma (line 33) and when checking whether Bad is blocked (line 59). This ensures any inductive invariant RACER finds is an $\mathcal{A}$ -solution to C_{fF}^k . RACER employs both uninterpreted function abstraction and k -instantiation abstraction when computing predecessors (line 54) and successors (line 48). This ensures that if Bad is reachable, it is reachable in $C_{fF}^{k\uparrow}$.

MBP RACER generates predecessor POBs by using MBP to eliminate post state variables. While there are algorithms to compute MBP for the theory of ADTs [15], it is not clear how to compute MBP in the presence of UF because of lack of quantifier elimination. The abstractions ensure that there are no UF when MBP is applied. Thus, RACER uses MBP for ADT+LIA from [15].

Progress From the properties of IC3, it follows that, whenever RACER or SPACER increments the depth of exploration (N in Alg. 8), there are no counterexamples of lower depth. We say that RACER (SPACER) makes *progress* if all non-terminating executions of RACER (SPACER) increment the depth of exploration infinitely often. In RACER (Alg. 8), this amounts to executing line 15 infinitely often on all non-terminating executions.

SPACER makes progress as long as the underlying theory is decidable and has a finite MBP. RACER keeps the underlying theory decidable by using abstractions to ensure that all queries are quantifier free formulas without RDFs. However, unlike SPACER, RACER does not use the same formula to check reachability of a POB and to compute its predecessors. Instead, RACER uses only the k -instantiation abstraction when checking reachability and uses both k -instantiation abstraction as well as uninterpreted function abstraction when computing predecessors using MBP. This makes arguing about progress a little tricky.

We first make two assumptions and show that RACER makes progress under those assumptions. We then relax those assumptions and argue why RACER still makes progress. Assume that k is fixed and that, during the one-step reachability check RACER employs both uninterpreted function abstraction as well as the k -instantiation abstraction instead of doing only the k -instantiation abstraction. Therefore, just like in the original SPACER, the formula to check reachability of a POB and the formula to compute its predecessors are the same. Since the MBP for the underlying theory is finite [15], each POB can only have a finite number of predecessors. Therefore, RACER makes progress under these assumptions. Now we relax the second assumption and employ *only* the k -instantiation abstraction during one-step reachability check. Let $\alpha_{rf}^k(\theta)$ be the k -instantiation abstracted formula that RACER uses to check reachability. We know that $\alpha_{rf}^{k\uparrow}(\theta)$ is weaker than $\alpha_{rf}^k(\theta)$. Therefore, all models of $\alpha_{rf}^k(\theta)$ are also models of $\alpha_{rf}^{k\uparrow}(\theta)$. Hence, when RACER computes MBP, no new models have been introduced just because we relaxed this assumption. Therefore, MBP is still be finite and RACER still makes progress. As for the first assumption, RACER makes progress as long as k is not increased too frequently. In our implementation, k is increased by 1 each time N is increased. Since the depth of counterexamples in any unsatisfiable CHCs is finite, the progress guarantee ensures that RACER is guaranteed to terminate on all unsatisfiable CHCs. Combining this with the results from earlier sections, we conclude the following:

Theorem 6 *Let C_{fF} be a set of relltionified CHCs modulo function axiom $\varphi_{(\mathbb{b}, \oplus)}$. If C_{fF} is UNSAT, RACER terminates and returns UNSAT. If C_{fF} is satisfiable, if RACER terminates, it returns a solution to C_{fF} . $\square$*

Category	Count	Eldarica	SPACER	HoICE	RACER
Programs with RDFs	44	0/0	1/8	18/3	36/0
Programs without RDFs	43	7/7	22/20	13/15	22/20
Total	87	7/7	23/28	31/18	58/20

Table 3.1: Comparing solved instances on RUST-BENCH. In each cell, the first entry is the number of SAT instances solved and the second entry is the number of UNSAT instances solved in that category.

Category	Count	Eldarica	VeriMAP	CVC4	SPACER	RACER
Queue	11	2	4 (2)	3	0	4 (1)
Tree	18	0	12 (1)	7 (1)	0	7
Heap	12	0	7 (3)	1	0	4 (1)
UNSAT	23	20	13	0	23 (1)	22
Total	64	22	36	11	23	37

Table 3.2: Comparing RACER with SPACER, Eldarica, CVC4, and VeriMAP on LEON-BENCH. The first three rows show SAT instances. The **UNSAT** row shows UNSAT instances from all three categories. The number of unique instances solved are shown in brackets.

3.7 Implementation and Evaluation

We have implemented RACER on top of SPACER in Z3³. Our implementation supports CHCs modulo multiple ADTs as well as multiple RDFs as long as all RDFs are relationified (i.e., in the form Eq. (3.6)).

Benchmark description While we are mainly interested in solving CHCs encoding imperative programs, to show the robustness of our approach, we evaluate RACER on CHC benchmarks obtained from two sources. The first group, called RUST-BENCH, contains 87 CHCs encoding safety of imperative Rust programs from [79]. The second

³Our implementation is available at <https://github.com/hgvk94/z3/tree/racer>

group, called LEON-BENCH, contains 64 CHCs encoding safety of ADT manipulating functional programs [28].

Getting benchmarks into the desired form While the benchmarks encode programs manipulating ADTs, they do not take advantage of RDFs. That is, the CHCs in these benchmarks capture programs using only uninterpreted predicates. They are relationified CHCs conjoined with positive relation constraints of RDFs (of the form Eq. (3.3), extended to include multiple F predicates).

We convert these CHCs into the desired form⁴ of C_{fF} (Eq. (3.6)) modulo all function axioms by identifying some predicates F with the RDFs f they correspond to, and recovering their RDF definitions. Specifically, we identify a subset of program methods as implementing RDFs and all other program methods (including recursive ones) as an implementation of some imperative program. For all program methods that define RDFs, we produce both their CHC encoding (positive relation constraint) and RDF axioms. All program methods that compute arithmetic properties of ADTs are considered as RDFs. For example, a method that checks whether a given list contains an element is considered an RDF, whereas a method that appends a given element to a list is not.

We use *term abstraction* to facilitate solutions that do not contain RDF applications. Specifically, we introduce new (arithmetic) variables to represent applications of RDFs that are necessary to prove safety. These variables are added to predicates in the CHCs as arguments that can be used in solutions. The equalities between the variables and the RDF applications that they represent are added to the bodies of the CHCs. Term abstraction is done greedily: for each RDF and each predicate argument, if the RDF can be applied either to the argument itself or, if the argument is a record type, to a selector applied to the argument, a new variable is introduced to capture the RDF application. We stress that this process is completely automatic. That is, we use the same exact input for all the tools in our evaluation. For our tool, we do a simple preprocessing step that identifies relationified RDFs, re-encodes them as RDFs, and generates terms for term abstraction.

Evaluation on Rust-bench The RUST-BENCH benchmark suite encodes Rust programs that manipulate ADTs. Of the 87 benchmarks, only 44 manipulate recursive ADTs (lists and trees) and contain (recovered) RDFs. The RDFs computed the length of lists, size of trees and sum of elements of trees and lists. On these benchmarks,

⁴Both modified and unmodified benchmarks are publicly available at <https://doi.org/10.5281/zenodo.5083780>.

we compare RACER with Eldarica⁵ [61], SPACER, and HOICE—state-of-the-art tools that support CHCs modulo ADT. Since none of these solvers support CHCs modulo RDFs, they cannot be run on our modified benchmarks. To make a comparison, we run them on the original set of benchmarks from [79]. We ran all four solvers with a timeout of 100s. Our results are shown in Tab. 3.1. In total, RACER solves 29 (resp. 27) more benchmarks than HOICE (resp. SPACER). When we only consider programs that contain RDFs, RACER solves 15 (resp. 27) more instances than HOICE (resp. SPACER). The superiority of RACER is clear when looking at the number of SAT instances it solves (58) compared to HOICE (31) and SPACER (23).

Evaluation on Leon-bench The benchmarks in LEON-BENCH encode verification condition validation queries generated by Leon [97]. The benchmarks encode operations on three types of ADTs: Queues, Trees, and Heaps. The RDFs compute size of the ADTs, checks membership in them, removes the last element, and checks whether they are sorted. We present our evaluation results on SAT and UNSAT CHCs separately. We further split the SAT benchmarks based on the type of ADTs.

On LEON-BENCH, we compare RACER with three CHC solvers: VeriMap [28], SPACER, Eldarica and with CVC4 [92] (with the subgoal generation heuristic enabled). For our evaluation, we use three different versions of these benchmarks: (1) to evaluate Verimap, Eldarica, and SPACER, we use the CHC encoding over ADTs and LIA, (2) to evaluate CVC4, we use the original verification condition validation queries but without any of the subgoals, as done in [28], and (3) to evaluate RACER, we conjoin RDFs to CHCs as mentioned earlier.

Tab. 3.2 shows the results of our evaluation. In summary, RACER is quite competitive with all other solvers even though RACER was designed for verifying imperative programs and not functional programs. In fact, in almost all categories, RACER solves benchmarks that other solvers do not solve.

We could not compare RACER with all solvers on all benchmarks because the tools do not support exactly the same inputs: Eldarica, HOICE, and SPACER support CHCs in the SMT2-LIB format, VeriMap supports CHCs modulo ADTs but in the Prolog format, and the inductive reasoning in CVC4 works with SMT-LIB formulas. We also repeated the experiments with a larger timeout of 30 minutes and did not see any change in number of instances solved for any of the solvers.

⁵We could not make use of Eldarica’s support for size constraints because in our benchmarks, RDFs are not restricted to size constraints.

3.8 Conclusion

In this chapter, we present a procedure for solving CHCs modulo ADTs and RDFs. Our approach sidesteps the undecidability of the underlying logic by simultaneously approximating the RDFs using multiple abstractions. One abstraction compiles RDF to CHCs while preserving satisfiability (but not solutions). It is used to detect unsatisfiability of CHCs (i.e., counterexamples to safety verification). The other abstraction, parameterized by the depth of unfolding k , replaces RDFs by their finite unfolding. As k increases, the abstraction enables more solutions that are potentially lost by the first abstraction. The two abstractions are explored in tandem in an IC3-style algorithm. Remarkably, the algorithm is able to automatically combine learning inductive invariants over ADTs and RDFs with learning inductive lemmas of RDFs.

While our presentation focuses on specific RDFs – catamorphisms – our results are more general. They apply to arbitrary well defined RDFs. The only requirements are that the function is total and it is possible to isolate the base- and recursive-cases in the definition.

We have implemented our technique on top of SPACER. Our evaluation shows significant improvement of our technique over state-of-the-art CHC solvers on imperative programs while still being competitive on functional programs.

Chapter 4

Fast Approximations Of Quantifier Elimination

Quantifier Elimination (qelim) is used in many automated reasoning tasks including program synthesis [73], exist-forall solving [31], [32], quantified SMT [15], and Model Checking [71]. Complete qelim, even when possible, is computationally expensive, and solvers often approximate it. We call these approximations *quantifier reductions*, to separate them from qelim. The difference is that quantifier reduction might leave some free variables in the formula.

For example, Z3 performs quantifier reduction, called QELITE, by greedily substituting variables by definitions syntactically appearing in the formulas. While it is very useful, it is necessarily sensitive to the order in which variables are substituted and depends on definitions appearing explicitly in the formula. Even though it may seem that these shortcomings need to be tolerated to keep QELITE fast, we show that it is not actually the case; we propose an egraph-based algorithm, QEL, to perform fast quantifier reduction that is complete relative to some semantic properties of the formula. Egraph [88] (see Sec. 1.2.1) is a data structure that compactly represents infinitely many terms and their equivalence classes. It was initially proposed as a decision procedure for EUF [88] and used for theorem proving (e.g., SIMPLIFY [29]). Since then, the applications of egraphs have grown. Egraphs are now used as term rewrite systems in equality saturation [63], [99], for theory combination in SMT solvers [29], [89], and for term abstract domains in Abstract Interpretation [20], [43], [53]. Using egraphs for rewriting or other formula manipulations (like qelim) requires a special operation, called *extract*, that converts nodes in the egraph back into terms. Term extraction was not considered when egraphs were first designed [88]. As far as we know, extraction was first studied in the application of egraphs for compiler optimization. Specifically, equality saturation [63], [98] is an optimization technique

over egraphs that consists in populating an egraph with many equivalent terms inferred by applying rules. When the egraph is saturated, i.e., applying the rules has no effect, the equivalent term that is most desired, e.g., smallest in size, is *extracted*. This is a recursive process that extracts each sub-term by choosing one representative among its equivalents.

Application of egraphs to rewriting have recently resurged driven by the `egg` library [103] and the associated workshop¹. In [103], the authors show, once again, the power and versatility of this data structure. Motivated by applications of equality saturation, they provide a generic and efficient framework equipped with term extraction, based on an extensible class analysis.

Egraphs seem to be the perfect data-structure to address the challenges of quantifier reduction: they allow reasoning about infinitely many equivalent terms and consider all available variable definitions and orderings at once. However, things are not always what they appear. The key to quantifier reduction is finding ground (i.e., variable-free) representatives for equivalence classes with free variables. This goes against existing techniques for term extraction since it requires selecting larger, rather than smaller, terms to be representatives. Selecting representatives carelessly makes term extraction diverge. To our surprise, this problem has not been studied so far. In fact, `egg` [103] incorrectly claims that any representative function can be used with its term extraction, while the implementation diverges. We bridge this gap by providing necessary and sufficient conditions for a representative function to be admissible for term extraction as defined in [63], [103]. Furthermore, we extend extraction from terms to formulas to enable extracting a formula of the egraph.

Our main contribution is a new quantifier reduction algorithm, called QEL. Building on the term extraction described above, it is formulated as finding a representative function that maximizes the number of ground terms as representatives. Furthermore, it greedily attempts to represent variables without ground representatives in terms of other variables, thus further reducing the number of variables in the output. We show that QEL is complete relative to ground definitions entailed by the formula. Specifically, QEL guarantees to eliminate a variable if it is equivalent to a ground term.

Whenever an application requires eliminating all free variables, incomplete techniques such as QELITE or QEL are insufficient. In this case, `qelim` is under-approximated using a Model Based Projection (MBP, see Sec. 1.3) that uses a model M of a formula to guide under-approximation using equalities and variable definitions that are consistent with M . We show that MBP for Arrays and Algebraic DataTypes (ADTs) can be implemented using our new techniques for QEL together with the machinery from equality saturation. in Sec. 4.4. Just like SMT solvers use

¹<https://pldi22.sigplan.org/series/egraphs>.

egraphs as glue to combine different theory solvers, we use egraphs as glue to combine projection for Arrays and ADTs. The algorithm uses insights from QEL to produce less under-approximate MBPs.

We implemented QEL and the new MBP using egraphs inside the state-of-art SMT solver **Z3 z3**. Our implementation (referred to as **Z3EG**) replaces the existing **QELITE** and **MBP**. We evaluate our algorithms in two contexts. First, inside the **QSAT** [15] algorithm for quantified satisfiability in the LA and ADT fragment. The performance of **QSAT** in **Z3EG** is improved, compared to **QSAT** in **Z3**, when ADTs are involved. Second, we evaluate our algorithms inside the Constrained Horn Clause (CHC) solver **SPACER** [71]. Our experiments show that **SPACER** in **Z3EG** solves many more benchmarks containing nested Arrays and ADTs.

Related Work. Quantifier reduction by variable substitution is widely used in quantified SMT [15], [48]. To our knowledge, we are the first to look at this problem semantically and provide an algorithm that guarantees that the variable is eliminated if the formula entails that it has a ground definition.

Term extraction for egraphs comes from equality saturation [63], [98]. The **egg** Rust library [103] is a recent implementation of equality saturation that supports rewriting and term extraction. However, we did not use **egg** because we integrated QEL within **Z3** and built it using **Z3** data structures instead.

Model Based Projection was first introduced for the **SPACER** CHC solver for LIA and LRA [71] and extended to the theory of Arrays [70] and ADTs [15]. Until now, it was implemented by syntactic rewriting. Our egraph-based MBP implementation is less sensitive to syntax and, more importantly, allows for combining MBPs of multiple theories for MBP of the combination. As a result, our MBP is more general and less model dependent. Specifically, it requires fewer model equalities and produces more general under-approximations than [15], [70].

Outline. The rest of the chapter is organized as follows. In Sec. 4.1, we explain how we construct an egraph of a formula. Sec. 4.2 introduces term extraction, extends it to formulas, and characterizes representative-based term extraction for egraphs. Sec. 4.3 presents QEL, our algorithm for fast quantifier reduction that is relatively complete. Sec. 4.4 shows how to compute MBP combining equality saturation and the ideas from Sec. 4.3 for the theories of ADTs and Arrays. All algorithms have been implemented in **Z3** and evaluated in Sec. 4.5.

This chapter has been published in the 35th International Conference on Computer Aided Verification (CAV 2023) [45].

$$\varphi_{4.1}(x, y, z) \triangleq z \approx \text{read}(a, x) \wedge k + 1 \approx \text{read}(a, y) \wedge x \approx y \wedge 3 > z$$

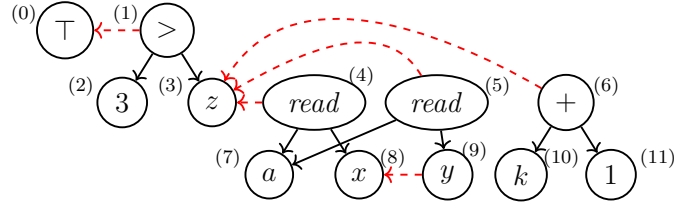
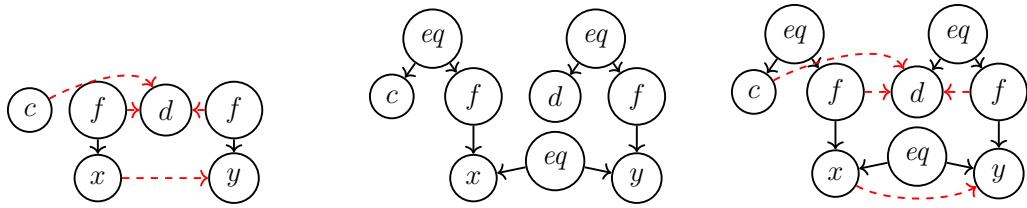


Figure 4.1: Example egraph of $\varphi_{4.1}$.

$$\varphi_{4.2}(x, y) \triangleq \text{eq}(c, f(x)) \wedge \text{eq}(d, f(y)) \wedge \text{eq}(x, y)$$



(a) G_a , interpreting eq as $\approx$. (b) G_b , not interpreting eq . (c) G_c , combining (a) and (b).

Figure 4.2: Different egraph interpretations for $\varphi_{4.2}$.

4.1 An Egraph of a Formula

We consider formulas that are conjunctions of equality literals (recall that we represent predicate applications by equality literals). Given a formula $\varphi \triangleq (t_1 \approx u_1 \wedge \dots \wedge t_k \approx u_k)$, an egraph from φ is built (following the standard procedure [88]) by creating nodes for each t_i and u_i , recursively creating nodes for their subexpressions, and merging the classes of each pair t_i and u_i , computing the congruence closure for **root**. We write $\text{egraph}(\varphi)$ for an egraph of φ constructed via some deterministic procedure based on the recipe above. Fig. 4.1 shows an $\text{egraph}(\varphi_1)$ of $\varphi_{4.1}$. The equality $z \approx \text{read}(a, x)$ is captured by $\mathbf{N}(3)$ and $\mathbf{N}(4)$ belonging to the same class (i.e., red arrow from $\mathbf{N}(4)$ to $\mathbf{N}(3)$). Similarly, the equality $x \approx y$ is captured by a red arrow from $\mathbf{N}(9)$ to $\mathbf{N}(8)$. Note that by congruence, $\varphi_{4.1}$ implies $\text{read}(a, x) \approx \text{read}(a, y)$, which, by transitivity, implies that $k + 1 \approx \text{read}(a, x)$. In Fig. 4.1, this corresponds to red arrows from $\mathbf{N}(5)$ and $\mathbf{N}(6)$ to $\mathbf{N}(3)$. The predicate application $3 > z$ is captured by the red arrow from $\mathbf{N}(1)$ to $\mathbf{N}(0)$. From now on, we omit $\top$ and $\perp$ and the corresponding edges from figures to avoid clutter.

Explicit and Implicit Equality. Note that egraphs represent equality implicitly by placing nodes with equal terms in the same equivalence class. Sometimes, it

is necessary to represent equality explicitly, for example, when using egraphs for equality-aware rewriting (e.g., in `egg` [103]). To represent equality explicitly, we introduce a binary `Bool` function `eq` and write $eq(a, b)$ for an equality that has to be represented explicitly. We change the *egraph* algorithm to treat $eq(a, b)$ as both a function application, and as a logical equality $a \approx b$: when processing term $eq(a, b)$, the algorithm both adds $eq(a, b)$ to the egraph, and merges the nodes for a and b into one class. For example, Fig. 4.2 shows three different interpretations of a formula $\varphi_{4.2}$ with equality interpreted: implicitly (as in [88]), explicitly (as in [103]), and both implicitly and explicitly (as in this work).

4.2 Extracting Formulas from Egraphs

Egraphs were proposed as a decision procedure for EUF [88] – a setting in which converting an egraph back to a formula, or *extracting*, is irrelevant. Term extraction has been studied in the context of equality saturation and term rewriting [63], [103]. However, existing literature presents extraction as a heuristic, and, to the best of our knowledge, has not been exhaustively explored. In this section, we fill these gaps in the literature and extend extraction from terms to formulas.

Term Extraction. We begin by recalling how to extract the term of a node. The function `ntt` (node-to-term) in Alg. 9 does an extraction parametrized by a representative function $\text{repr} : N \mapsto N$ (same as in [103]). A function `repr` assigns each class a unique representative node (i.e., nodes in the same class are mapped to the same representative) so that $\rho_{\text{root}} = \rho_{\text{repr}}$. The function `ntt` extracts a term of a node recursively, similarly to *term*, except that the representatives of the children of a node are used instead of the actual children. We refer to terms built in this way by `ntt(n, repr)` and omit `repr` when it is clear from the context.

As an example, consider $\text{repr}_1 \triangleq \{\mathbf{N}(3), \mathbf{N}(8)\}$ for Fig. 1.1. For readability, we denote representative functions by sets of nodes that are the class representatives, omitting $\mathbf{N}(\top)$ that always represents its class, and omitting all singleton classes. Thus, repr_1 maps all nodes in $\text{class}(\mathbf{N}(3))$ to $\mathbf{N}(3)$, nodes in $\text{class}(\mathbf{N}(8))$ to $\mathbf{N}(8)$, nodes in $\text{class}(\mathbf{N}(\top))$ to $\mathbf{N}(\top)$, and all singleton classes to themselves. For example, `ntt(N(5))` extracts $\text{read}(a, x)$, since $\mathbf{N}(9)$ has as representative $\mathbf{N}(8)$.

Formula Extraction. Let $G = \text{egraph}(\varphi)$ be an egraph of some formula φ . A formula ψ is a *formula of G*, written $\text{isFormula}(G, \psi)$, if $\psi^\exists \equiv \varphi^\exists$.

Alg. 9 shows an algorithm `to_formula(repr, S)` to compute a formula ψ that satisfies $\text{isFormula}(G, \psi)$ for a given egraph G . In addition to `repr`, `to_formula`

Algorithm 9: Producing formulas from an egraph.

<pre> 1 function <i>egraph</i> :: to_formula(repr, S): 2 <i>Lits</i> := $\emptyset$ 3 for $r = \text{repr}(r) \in N$ do 4 $t := \text{ntt}(r, \text{repr})$ 5 for $n \in (\text{class}(r) \setminus \{r\})$ do 6 if $n \notin S$ then 7 <i>Lits</i> := <i>Lits</i> $\cup \{t \approx \text{ntt}(n, \text{repr})\}$ 8 return $\bigwedge \text{Lits}$ </pre>	<pre> 9 function <i>egraph</i> :: ntt(n, repr): 10 <i>egraph</i> :: ntt(n, repr) 11 $f := L[n]$ 12 if $\text{deg}(n) = 0$ then 13 return f 14 else 15 for $i \in [1, \text{deg}(n)]$ do 16 $\text{Args}[i] := \text{ntt}(\text{repr}(n[i]), \text{repr})$ 17 return $f(\text{Args})$ </pre>
---	---

is parameterized by a set of nodes $S \subseteq N$ to exclude². To produce the equalities corresponding to the classes, for each representative r , for each $n \in (\text{class}(r) \setminus \{r\})$ the output formula has a literal $\text{ntt}(r) \approx \text{ntt}(n)$. For example, using repr_1 for the egraph in Fig. 1.1, we obtain for $\text{class}(\mathbf{N}(8))$, $(x \approx y)$; for $\text{class}(\mathbf{N}(3))$, $(z \approx \text{read}(a, x) \wedge z \approx \text{read}(a, x) \wedge z \approx k + 1)$; and for $\text{class}(\mathbf{N}(0))$, $(\top \approx 3 > z)$. The final result (slightly simplified) is: $x \approx y \wedge z \approx \text{read}(a, x) \wedge z \approx k + 1 \wedge 3 > z$.

Let $G = \text{egraph}(\varphi)$ for some formula φ . Note that, ψ computed by `to_formula` is not syntactically the same as φ . That is, `to_formula` is not an inverse of `egraph`. Furthermore, since `to_formula` commits to one representative per class, it is limited in what formulas it can generate. For example, since $x \approx y$ is in $\varphi_{1.1}$, for any repr , $\varphi_{1.1}$ cannot be the result of `to_formula`, because the output can contain only one of $\text{read}(a, x)$ or $\text{read}(a, y)$.

Representative Functions. The representative function is instrumental for determining the terms that appear in the extracted formula. To illustrate the importance of representative choice, consider the formula $\varphi_{4.3}$ of Fig. 4.3 and its egraph $G_4 = \text{egraph}(\varphi_{4.3})$. For now, ignore the blue dotted lines. For $\text{repr}_{4.3a}$, `to_formula` obtains $\psi_a \triangleq (x \approx g(6) \wedge f(x) \approx 6 \wedge y \approx 6)$. For $\text{repr}_{4.3b}$, `to_formula` produces $\psi_b \triangleq (g(6) \approx x \wedge f(g(6)) \approx 6 \wedge y \approx 6)$. In some applications (like qelim considered in this work) ψ_b is preferred to ψ_a : simply removing the literals $g(6) \approx x$ and $y \approx 6$ from ψ_b results in a formula equivalent to $\exists x, y \cdot \varphi_{4.3}$ that does not contain variables. Consider a third representative choice $\text{repr}_{4.3c}$, for node $\mathbf{N}(1)$, `ntt` does not terminate: to produce a term for $\mathbf{N}(1)$, a term for $\mathbf{N}(3)$, the representative of its child, $\mathbf{N}(2)$, is required. Similarly to produce a term for $\mathbf{N}(3)$, a term for the representative of its child node $\mathbf{N}(5)$, $\mathbf{N}(1)$, is necessary. Thus, none of the terms can be extracted with $\text{repr}_{4.3c}$.

For extraction, representative functions `repr` are either provided explicitly or

²The set S affects the result, but for this section, we restrict to the case of $S \triangleq \emptyset$.

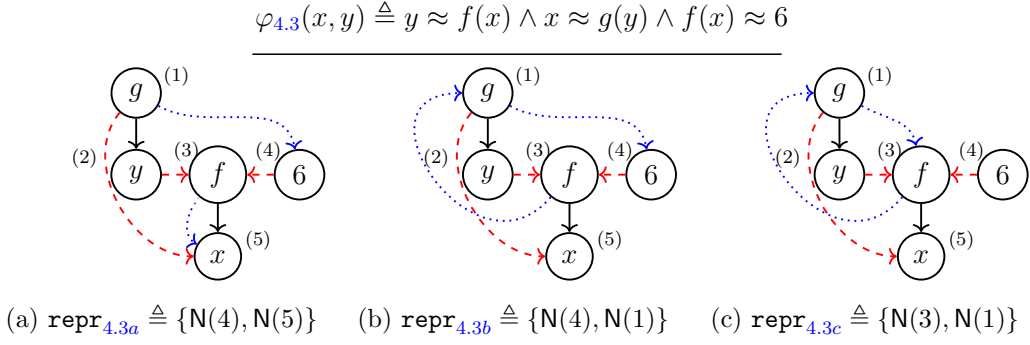


Figure 4.3: Egraphs of $\varphi_{4.3}$ with G_{repr} .

implicitly (as in [103]), the latter by associating a cost to nodes and/or terms and letting the representative be a node with minimal cost. However, observe that not all costs guarantee that the chosen repr can be used (the computation does not terminate). For example, the ill-defined $\text{repr}_{4.3c}$ from above is a representative function that satisfies the cost function that assigns function applications cost 0 and variables and constants cost 1. A commonly used cost function is term AST size, which is sufficient to ensure termination of $\text{ntt}(n, \text{repr})$.

We are thus interested in characterizing representative functions motivated by two observations: not every cost function guarantees that $\text{ntt}(n)$ terminates; and the kind of representative choices that are most suitable for qelim ($\text{repr}_{4.3b}$) cannot be expressed over term AST size.

Definition 9 Given an egraph $G = \langle N, E, L, \text{root} \rangle$, a representative function $\text{repr} : N \rightarrow N$ is *admissible* for G if

- (a) repr assigns a unique representative per class,
- (b) $\rho_{\text{root}} = \rho_{\text{repr}}$, and
- (c) the graph G_{repr} is acyclic, where $G_{\text{repr}} = \langle N, E_{\text{repr}} \rangle$ and $E_{\text{repr}} \triangleq \{(n, \text{repr}(c)) \mid c \in \text{children}(n), n \in N\}$. $\square$

Dotted blue edges in the graphs of Fig. 4.3 show the corresponding G_{repr} . Intuitively, for each node n , all reachable nodes in G_{repr} are the nodes whose ntt term is necessary to produce the $\text{ntt}(n)$. Observe that $G_{\text{repr}_{4.3c}}$ has a cycle, thus, $\text{repr}_{4.3c}$ is not admissible. In the rest of this section, we formally show that admissibility is a necessary and sufficient condition for termination of `to_formula`.

Assume that $G = \langle N, E, L, \text{root} \rangle$ is an egraph, $n, m \in N$ are nodes in G , repr is a representative function for G , and that $G_{\text{repr}} = \langle N, E_{\text{repr}} \rangle$ is the graph in the

Def. 9. We define the *execution trace* of an execution of a procedure as the sequence of procedure calls made during the execution.

Lemma 4 $\text{ntt}(n_k, \text{repr})$ is in the execution trace of $\text{ntt}(n, \text{repr})$ iff there is a path from n to n_k in G_{repr} . $\square$

PROOF *Only if direction.* By induction on the length of the path between n and n_k in G_{repr} .

Base case. If the length is one, $n_k = \text{repr}(n[1])$, the representative of the child of n . Since n can not be of degree zero, $\text{ntt}(n, \text{repr})$ reaches line 16 and recurses on the representatives of all children of n . Since n_k is one of them, $\text{ntt}(n_k, \text{repr})$ is in the execution trace of $\text{ntt}(n, \text{repr})$.

Inductive step. Assume that n_j is reachable from n in G_{repr} and that execution trace of $\text{ntt}(n, \text{repr})$ contains $\text{ntt}(n_j, \text{repr})$. We prove that, if there exists an edge (n_j, n_{j+1}) in G_{repr} , execution trace of $\text{ntt}(n, \text{repr})$ contains $\text{ntt}(n_{j+1}, \text{repr})$. Since there is an edge (n_j, n_{j+1}) , n_{j+1} is the representative of one of the children of n_j . Therefore, $\text{ntt}(n_j, \text{repr})$ reaches line 16 and calls $\text{ntt}(n_{j+1}, \text{repr})$. Since the execution trace of $\text{ntt}(n, \text{repr})$ contains $\text{ntt}(n_j, \text{repr})$, it must also contain $\text{ntt}(n_{j+1}, \text{repr})$.

If direction. We prove it by induction on the length of an execution trace.

Base case. The execution trace of $\text{ntt}(n, \text{repr})$ is of length 1 only if n has exactly one child. Therefore, $n_k = n[1]$ is the representative of the only child of n . By Def. 9, $(n, n_k) \in E_{\text{repr}}$.

Inductive step. Our inductive hypothesis is that for all j s.t. $1 < j \leq i$, $\text{ntt}(n_j, \text{repr})$ is in the execution trace and there is a path from n to n_j . We have to show that, if $\text{ntt}(n_{i+1}, \text{repr})$ is in the execution trace, there is a path from n to n_{i+1} . Since Alg. 9 recurses only on the representatives of children of a node (line 16), $\text{ntt}(n_{i+1}, \text{repr})$ is in the trace only if n_{i+1} is the representative of a child of a node n_j where $1 \leq j \leq i$. By our inductive hypothesis, either $j = 1$ or there is already a path from n to n_j . Since n_{i+1} is the representative of a child of n_j , by Def. 9, there is an edge from n_j to n_{i+1} . Hence, there is a path from n to n_{i+1} . $\blacksquare$

Lemma 5 If there is a path from n to n_k in G_{repr} , then $n \neq \text{repr}(n_k)$. $\square$

PROOF Let the path from n to n_k be $n_1 = n, n_2, \dots, n_k$ where $\forall 1 \leq j < k \cdot \exists i \cdot n_{j+1} = \text{repr}(n_j[i])$. For contradiction, assume that $n = \text{repr}(n_k)$. By Def. 9(a), each equivalence class has exactly one representative. Therefore $\text{repr}(n_k) = \text{repr}(n) = n$. The path $n, \dots, n_{k-1}, n$ forms a cycle in G_{repr} . $\blacksquare$

Corollary 2 The maximum length of a path in G_{repr} is the number of equivalence classes in G . $\square$

Lemma 6 *Let M and d respectively be the number of equivalence classes and maximum out degree in G . The complexity of $\text{ntt}(n, \text{repr})$ is $\mathcal{O}(M^d)$.* $\square$

PROOF Let $T(n)$ denote the time to run $\text{ntt}(n, \text{repr})$. $T(n) = (\sum_{i=1, \dots, d} T(n[i])) + \mathcal{O}(1)$. This forms a tree with height M , the maximum length of the path in G_{repr} . The number of leaf nodes is M^d . $\blacksquare$

Lem. 6 states that acyclicity of G_{repr} is a sufficient condition for termination of `to_formula`. The following lemma states that it is also a necessary condition.

Lemma 7 *For a node $n \in N$, let p be the longest path starting from n in G_{repr} . The complexity of $\text{ntt}(n, \text{repr})$ is $\Omega(\text{length}(p))$.* $\square$

PROOF By Lem. 4, if there is a path from n to some leaf n_l , $(n, \dots, n_i, \dots, n_l)$, all n_i are on the execution trace of $\text{ntt}(n, \text{repr})$. $\blacksquare$

From Lem. 7, it follows that if G_{repr} has a cycle involving some node n , $\text{ntt}(n, \text{repr})$ does not terminate, since there is no bound on the longest path, the execution trace does not have a bound either.

Lemma 8 *Given a formula φ and $G = \text{egraph}(\varphi)$ with an admissible representative function repr . For every $n \in G$, $\varphi \models (\text{term}(n) \approx \text{ntt}(n, \text{repr}))$.* $\square$

PROOF By Cor. 2, for every node n , $\text{ntt}(n, \text{repr})$ is defined.

We prove by induction. For readability, we omit the repr parameter of ntt . For the base case, if n has no children $\text{term}(n)$ is exactly $\text{ntt}(n)$. Therefore, by condition (b) of admissibility of repr , $\varphi \models \text{term}(n) \approx \text{term}(\text{repr}(n))$.

For the inductive case, let $f = L(n)$. For every child $n[i]$:
 By hypothesis, $\varphi \models (\text{term}(\text{repr}(n[i])) \approx \text{ntt}(\text{repr}(n[i])))$;
 by transitivity, $\varphi \models \text{term}(n[i]) \approx \text{ntt}(\text{repr}(n[i]))$;
 by congruence, $\varphi \models (f(\text{term}(n[i])) \approx f(\text{ntt}(\text{repr}(n[i]))))$;
 by definition, $f(\text{term}(n[i])) = \text{term}(n)$ and $f(\text{ntt}(n[i])) = \text{ntt}(n)$.
 Therefore $\varphi \models (\text{term}(n) \approx \text{ntt}(n))$. $\blacksquare$

Finally, we are ready to prove that admissibility is necessary and sufficient for termination:

Theorem 7 *Given an egraph G and a representative function repr , the function $G.\text{to_formula}(\text{repr}, \emptyset)$ terminates with result ψ such that $\text{isFormula}(G, \psi)$ iff repr is admissible for G .* $\square$

PROOF Termination in both directions follows from Lem. 6 and 7.

Given a formula φ , and its egraph $G = \text{egraph}(\varphi)$, and admissible repr , let $\psi = G.\text{to_formula}(\text{repr}, \emptyset)$ we show that $\text{isFormula}(G, \psi)$. We prove that $\varphi \leftrightarrow \psi$ is a valid formula. We do so in two steps. (1) Let ψ be of the form $(\ell_1, \dots, \ell_n)$ then $\varphi \models \ell_i$, and (2) let φ be of the form $(\ell_1, \dots, \ell_n)$ then $\psi \models \ell_i$.

Step (1), let ψ be of the form $(\ell_1, \dots, \ell_n)$ then $\varphi \models \ell_i$. Each ℓ_i is of the form $\text{ntt}(r) \approx \text{ntt}(n)$ where $r = \text{repr}(n)$. By construction and completeness of egraphs, and conditions (a) and (b) of Def. 9, $\varphi \models \text{term}(r) \approx \text{term}(n)$. By Lem. 8, $\varphi \models \text{term}(r) \approx \text{ntt}(r)$ and $\varphi \models \text{term}(n) \approx \text{ntt}(n)$. Then, by transitivity, $\varphi \models (\text{ntt}(r) \approx \text{ntt}(n))$.

Step (2), let φ be of the form $(\ell_1, \dots, \ell_n)$ then $\psi \models \ell_i$. Each ℓ_i is of the form $\text{term}(n) \approx \text{term}(m)$ where n and m are nodes in G . ψ contains literals $\text{ntt}(\text{repr}(n)) \approx \text{ntt}(n)$ and $\text{ntt}(\text{repr}(m)) \approx \text{ntt}(m)$, then by transitivity we have $\psi \models \text{ntt}(n) \approx \text{ntt}(m)$. Without loss of generality, assume that $n = \text{repr}(m)$, then $\psi \models \text{ntt}(n) \approx \text{ntt}(m)$. We prove that $\psi \models \text{ntt}(n) \approx \text{term}(n)$, and the previous equality will follow by transitivity. By induction. For the base case, n has no children $\text{ntt}(n)$ is exactly $\text{term}(n)$, therefore, $\psi \models \text{ntt}(n) \approx \text{term}(n)$. For the inductive case, assume that for all children $n[i]$, $\psi \models \text{ntt}(n[i]) \approx \text{term}(n[i])$. $\psi \models \text{ntt}(\text{repr}(n)) \approx \text{ntt}(n)$, therefore $\psi \models \text{ntt}(\text{repr}(n[i])) \approx \text{ntt}(n[i])$. Then, by congruence and transitivity $\psi \models f(\text{ntt}(\text{repr}(n[i]))) \approx f(\text{term}(n[i]))$, and by definition $\psi \models \text{ntt}(n) \approx \text{term}(n)$. ■

To the best of our knowledge, theorem 7 is the first complete characterization of all terms of a node that can be obtained by extraction based on class representatives (via describing all admissible repr , note that the number is finite). This result contradicts [103], where it is claimed to be possible to extract a term of a node for any cost function. The counterexample is $\text{repr}_{4.3c}$. Importantly, this characterization allows us to explore representative functions outside those in the existing literature, which, as we show in the next section, is key for qelim.

4.3 Quantifier Reduction

Quantifier reduction is a relaxation of quantifier elimination: given two formulas φ and ψ with free variables $\bar{v}$ and $\bar{u}$, respectively, ψ is a *quantifier reduction* of φ if $\bar{u} \subseteq \bar{v}$ and $\varphi^\exists \equiv \psi^\exists$. If $\bar{u}$ is empty, then ψ is a quantifier elimination of $\varphi^\exists$. Note that quantifier reduction is possible even when quantifier elimination is not (e.g., for EUF). We are interested in an efficient quantifier reduction algorithm (that can be used as pre-processing for qelim), even if a complete qelim is possible (e.g., for LIA). In this section, we present such an algorithm called QEL.

Algorithm 10: QEL – Quantifier reduction using egraphs.

Input: A formula φ with free variables $\bar{v}$.

Output: A quantifier reduction of φ .

```
1 function  $QEL(\varphi, \bar{v})$ :  
2    $G := egraph(\varphi)$   
3    $repr := G.find\_defs(\bar{v})$   
4    $repr := G.refine\_defs(repr, \bar{v})$   
5    $core := G.find\_core(repr)$   
6   return  $G.to\_formula(repr, G.Nodes() \setminus core)$ 
```

Intuitively, QEL is based on the well-known substitution rule: $(\exists x \cdot x \approx t \wedge \varphi) \equiv \varphi[x \mapsto t]$. A naive implementation of this rule, called QELITE in Z3, looks for syntactic definitions of the form $x \approx t$ for a variable x and an x -free term t and substitutes x with t . While efficient, QELITE is limited because of: (a) dependence on syntactic equality in the formula (specifically, it misses implicit equalities due to transitivity and congruence); (b) sensitivity to the order in which variables are eliminated (eliminating one variable may affect available syntactic equalities for another); and (c) difficulty in dealing with circular equalities such as $x \approx f(x)$.

For example, consider the formula $\varphi_{4.3}(x, y)$ in Fig. 4.3. Assume that y is eliminated first using $y \approx f(x)$, resulting in $x \approx g(f(x)) \wedge f(x) \approx 6$. Now, x cannot be eliminated since the only equality for x is circular. Alternatively, assume that QELITE somehow noticed that by transitivity, $\varphi_{4.3}$ implies $y \approx 6$, and obtains $(\exists y \cdot \varphi_{4.3}) \triangleq x \approx g(6) \wedge f(x) \approx 6$. This time, $x \approx g(6)$ can be used to obtain $f(g(6)) \approx 6$ that is a qelim of $\varphi_{4.3}$. Thus, both the elimination order and implicit equalities are crucial.

In QEL, we address the above issues by using an egraph data structure to concisely capture all implicit equalities and terms. Furthermore, egraphs allow eliminating multiple variables together, ensuring that a variable is eliminated if it is equivalent (explicitly or implicitly) to a ground term in the egraph.

Pseudocode for QEL is shown in Alg. 10. Given an input formula φ , QEL first builds its egraph G (line 1). Then, it finds a representative function **repr** that maps variables to equivalent ground terms, as much as possible (line 2). Next, it further reduces the remaining free variables by refining **repr** to map each variable x to an equivalent x -free (but not variable-free) term (line 4). At this point, QEL is committed to the variables to eliminate. To produce the output, **find_core** identifies the subset of the nodes of G , which we call *core*, that must be considered in the output (line 5). Finally, **to_formula** converts the core of G to the resulting formula (line 6). We show that the combination of these steps is even stronger than variable substitution.

To illustrate QEL, we apply it on $\varphi_{1.1}$ and its egraph G from Fig. 1.1. The function

$$\varphi_{4.4}(x, y) \triangleq x \approx g(f(x)) \wedge y \approx h(f(y)) \wedge f(x) \approx f(y)$$

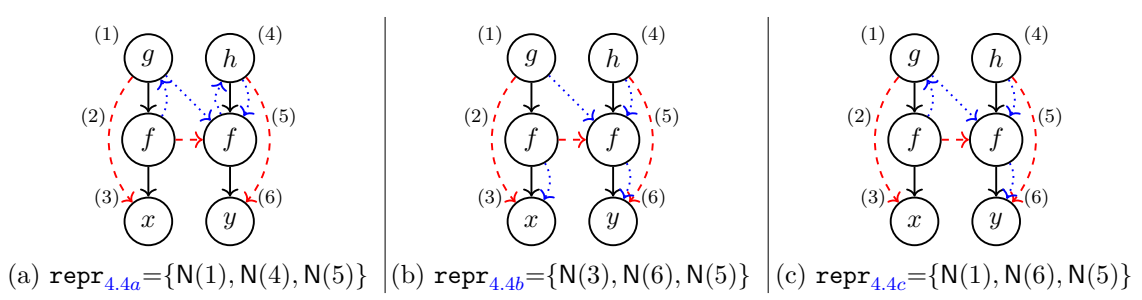


Figure 4.4: Egraphs including G_{repr} of $\varphi_{4.4}$.

`find_defs` returns $\text{repr} = \{N(6), N(8)\}$ ³. Node $N(6)$ is the only node with a ground term in the equivalence class $\text{class}(N(3))$. This corresponds to the definition $z \approx k + 1$. Node $N(8)$ is chosen arbitrarily since $\text{class}(N(8))$ has no ground terms. There is no refinement possible, so `refine_defs` returns repr . The core is $N \setminus \{N(3), N(5), N(9)\}$. Nodes $N(3)$ and $N(9)$ are omitted because they correspond to variables with definitions (under repr), and $N(5)$ is omitted because it is congruent to $N(4)$ so only one of them is needed. Finally, `to_formula` produces $k + 1 \approx \text{read}(a, x) \wedge 3 > k + 1$. Variables z and y are eliminated.

In the rest of this section we present QEL in detail and QEL’s key properties.

Finding Ground Definitions. Ground variable definitions are found by selecting a representative function repr that ensures that the maximum number of terms in the formula are rewritten into ground equivalent ones, which, in turn, means finding a ground definition for all variables that have one.

Computing a representative function repr that is admissible and ensures finding ground definitions when they exist is not trivial. Naive approaches for identifying ground terms, such as iterating arbitrarily over the classes and selecting a representative based on $\text{term}(n)$ are not enough – $\text{term}(n)$ may not be in the output formula. It is also not possible to make a choice based on $\text{ntt}(n)$, since, in general, it cannot be yet computed (repr is not known yet).

Admissibility raises an additional challenge since choosing a node that appears to be a definition (e.g., not a leaf) may cause cycles in G_{repr} . For example, consider $\varphi_{4.4}$ of Fig. 4.4. Assume that $N(1)$ and $N(4)$ are chosen as representatives of their equivalence classes. At this point, G_{repr} has two edges: $\langle N(5), N(4) \rangle$ and $\langle N(2), N(1) \rangle$, shown by blue dotted lines in Fig. 4.4a. Next, if either $N(2)$ or $N(5)$ are chosen as

³Recall that we only show representatives of non-singleton classes.

Algorithm 11: Find definitions maximizing groundness.

<pre> 1 function <i>egraph</i> :: find_defs($\bar{v}$): 2 for $n \in N$ do 3 $\text{repr}(n) := \star$ 4 $\text{todo} := \{\text{leaf}(n) \mid n \in N \wedge \text{ground}(n)\}$ 5 $\text{repr} := \text{process}(\text{repr}, \text{todo})$ 6 $\text{todo} := \{\text{leaf}(n) \mid n \in N\}$ 7 $\text{repr} := \text{process}(\text{repr}, \text{todo})$ 8 return repr </pre>	<pre> 9 function <i>egraph</i> :: process(repr, todo): 10 while $\text{todo} \neq \emptyset$ do 11 $n := \text{todo.pop}()$ 12 if $\text{repr}(n) \neq \star$ then 13 continue 14 for $n' \in \text{class}(n)$ do 15 $\text{repr}(n') := n$ 16 for $n' \in \text{class}(n)$ do 17 for $p \in \text{parents}(n')$ do 18 if $\forall c \in \text{children}(p) \cdot \text{repr}(c) \neq \star$ then 19 $\text{todo.push}(p)$ 20 return repr </pre>
--	--

representatives (the only choices in their class), then G_{repr} becomes cyclic (shown in blue in Fig. 4.4a). Furthermore, backtracking on representative choices needs to be avoided if we are to find a representative function efficiently.

Alg. 11 finds a representative function **repr** while overcoming these challenges. To ensure that the computed representative function is admissible (without backtracking), Alg. 11 selects representatives for each class using a “bottom up” approach. Namely, leaves cannot be part of cycles in G_{repr} because they have no outgoing edges. Thus, they can always be safely chosen as representatives. Similarly, a node whose children have already been assigned representatives in this way (leaves initially), will also never be part of a cycle in G_{repr} . Therefore, these nodes are also safe to be chosen as representatives.

This intuition is implemented in **find_defs** by initializing **repr** to be undefined ($\star$) for all nodes, and maintaining a workset, *todo*, containing nodes that, if chosen for the remaining classes (under the current selection), maintain acyclicity of G_{repr} . The initialization of *todo* includes leaves only. The specific choice of leaves ensures that ground definitions are preferred, and we return to it later. After initialization, the function **process** extracts an element from *todo* and sets it as the representative of its class if the class has not been assigned yet (lines 12 and 15). Once a class representative has been chosen, on lines 16 to 19, the parents of all the nodes in the class such that all the children have been chosen (the condition on line 18) are added to *todo*.

So far, we discussed how admissibility of **repr** is guaranteed. To also ensure that ground definitions are found whenever possible, we observe that a similar bottom up approach identifies terms that can be rewritten into ground ones. This builds on the notion of constructively ground nodes, defined next.

A class c is *ground* if c contains a *constructively ground*, or *c-ground* for short, node n , where a node n is c-ground if either (a) $\text{term}(n)$ is ground, or (b) n is not a leaf and the class $\text{class}(n[i])$ of every child $n[i]$ is ground. Note that nodes labeled by variables are never c-ground.

In the example in Fig. 1.1, $\text{class}(\mathbf{N}(7))$ and $\text{class}(\mathbf{N}(8))$ are not ground, because all their nodes represent variables; $\text{class}(\mathbf{N}(6))$ is ground because $\mathbf{N}(6)$ is c-ground. Nodes $\mathbf{N}(4)$ and $\mathbf{N}(5)$ are not c-ground because the class of $\mathbf{N}(8)$ (a child of both nodes) is not ground. Interestingly, $\mathbf{N}(1)$ is c-ground, because $\text{class}(\mathbf{N}(3)) = \text{class}(\mathbf{N}(6))$ is ground, even though its term $3 > z$ is not ground.

Ground classes and c-ground nodes are of interest because whenever $\varphi \models \text{term}(n) \approx t$ for some node n and ground term t , then $\text{class}(n)$ is ground, i.e., it contains a c-ground node, where c-ground nodes can be found recursively starting from ground leaves. Furthermore, the recursive definition ensures that when the aforementioned c-ground nodes are selected as representatives, the corresponding terms w.r.t. **repr** are ground.

As a result, to maximize the ground definitions found, we are interested in finding an admissible representative function **repr** that is *maximally ground*, which means that for every node $n \in N$, if $\text{class}(n)$ is ground, then **repr**(n) is c-ground. That means that c-ground nodes are always chosen if they exist.

We note that not every choice of c-ground nodes as representatives results in an admissible representative function. For example, consider the formula $\varphi_{4.3}$ of Fig. 4.3 and its egraph. All nodes except for $\mathbf{N}(5)$ and $\mathbf{N}(2)$ are c-ground. However, a **repr** with $\mathbf{N}(3)$ and $\mathbf{N}(1)$ as representatives is not admissible. Intuitively, this is because the “witness” for c-groundness of $\mathbf{N}(1)$ in $\text{class}(\mathbf{N}(2))$ is $\mathbf{N}(4)$ and not $\mathbf{N}(3)$. Therefore, it is important to incorporate the selection of c-ground representatives into the bottom up procedure that ensures admissibility of **repr**.

To promote c-ground nodes over non c-ground in the construction of an admissible representative function, **find_defs** chooses representatives in two steps. First, only the ground leaves are processed (line 4). This ensures that c-ground representatives are chosen while guaranteeing the absence of cycles. Then, the remaining leaves are added to *todo* (line 6). This triggers representative selection of the remaining classes (those that are not ground).

We illustrate **find_defs** with two examples. For $\varphi_{4.3}$ of Fig. 4.3, there is only one leaf that is ground, $\mathbf{N}(4)$, which is added to *todo* on line 4, and *todo* is processed. $\mathbf{N}(4)$ is chosen as representative and, as a consequence, its parent $\mathbf{N}(1)$ is added to *todo*. $\mathbf{N}(1)$ is chosen as representative so $\mathbf{N}(3)$, even though added to the queue later, is not chosen as representative, obtaining $\text{repr}_{4.3b} = \{\mathbf{N}(4), \mathbf{N}(1)\}$. For $\varphi_{4.4}$ of Fig. 4.4, no nodes are added to *todo* on line 4. $\mathbf{N}(3)$ and $\mathbf{N}(6)$ are added on line 6. In **process**, both are chosen as representatives obtaining, $\text{repr}_{4.4b}$.

In the rest of this section, we state and prove the correctness of Alg. 11. In the following, we use *partial* representative functions to mean representative functions with $\star$ in its range. For a partial representative function, we say that $\text{repr}(n)$ is *defined* if $\text{repr}(n) \neq \star$. Let $G = \langle N, E, L, \text{root} \rangle$ be an egraph and $\text{repr} = G.\text{find_defs}(\bar{v})$. We use r, r_1, r_2, r_i to denote partial representative functions, the intermediate results computed during the execution of `find_defs`.

The following lemma states that once a node is in the *todo* list, it will get assigned a representative.

Lemma 9 *For any $n \in N$, let $r_2 = \text{process}(\text{repr}, \text{todo})$ be such that $n \in \text{todo}$ during the execution of `process`. Then $r_2(n) \neq \star$* $\square$

PROOF Since `process` never removes representatives, if $\text{repr}(n) \neq \star$, $r_2(n) \neq \star$. If $n \in \text{todo}$, it is eventually processed. At this point, either $\text{repr}(n)$ is already defined, or n is chosen as the representative for its class. `process` terminates because in each iteration, it either decreases the number of classes without representative or, if the number of classes without representatives remains the same, it decreases the size of *todo*. $\blacksquare$

To prove that `process`(r, todo) assigns a representative to a node $n \in N$, we analyze the descendants of n . If all children of n are in *todo*, `process`(r, todo) assigns a representative to n . Furthermore, if $c \in \text{children}(n)$ is not in *todo* but all children of c are in *todo*, then, `process`(r, todo) assigns representative to c , adds c to *todo*, and then assigns representative to n . Thus, even if a descendant of n is not in *todo*, it is enough that all the children of the descendant are in *todo*. Such sets are called *frontiers* of a node. The set of children of n is one frontier of n . Given a frontier, replacing any non-leaf node with all of its children gives us another frontier. Formally, we define the set of all frontiers of n as:

Definition 10 (Frontiers of a node) Given a graph $G = \langle N, E \rangle$, the *frontiers* of a node $n \in N$, denoted $\mathcal{F}(n)$, is a set of sets of nodes defined as

$$\mathcal{F}(n) = \{\{\text{children}(n)\}\} \cup \{(\Pi \cup \{\text{children}(m)\}) \setminus \{m\} \mid \Pi \in \mathcal{F}(n) \wedge m \in \Pi \wedge \deg(m) > 0\} \quad \square$$

Lemma 10 *Let $r_2 = \text{process}(r_1, \text{todo})$. For any node $n \in N$ s.t. $\exists \Pi \in \mathcal{F}(n) \cdot \Pi \subseteq \text{todo}$, $r_2(n) \neq \star$.* $\square$

PROOF We prove it by induction on the frontiers of n . For the base case, $\Pi = \{\text{children}(n)\}$. `find_defs` assigns a representative to each $c \in \Pi$ at line 15, if they

did not have one. After assigning a representative for the last child, the condition on line 18 is true for n . Therefore, n is added to the *todo* and a representative for n is chosen (Lem. 9). For the inductive step, assume that the lemma holds for a set $\Pi \in \mathcal{F}(n)$. That is, if $\Pi \subseteq \text{todo}$, and $r_2 = \text{process}(r_1, \text{todo})$, $r_2(n)$ defined. We prove that the lemma holds for $\Pi' = (\Pi \cup \{\text{children}(m)\}) \setminus \{m\}$, where m is a non-leaf node in Π . Since all nodes in Π' are in *todo*, all these nodes will eventually have representatives (Lem. 9). When the representative of the last child is chosen, m is a node such that either it has a representative or all its children have representatives. In the second case, m is added to *todo* (line 18) and by Lem. 9, its representative is eventually picked. Hence, $r_2(m) \neq \star$. ■

Lemma 11 *For all $n \in N$, $\text{repr}(n)$ is defined.* □

PROOF Alg. 11 calls **process** with all leaf nodes. For any node n , there is a $\Pi \in \mathcal{F}(n)$ s.t. all nodes in Π are leaf. Therefore, by Lem. 10 $\text{repr}(n)$ is defined for all nodes. ■

Next, we prove that Alg. 11 picks a maximally ground admissible **repr**. For the proof, we extend the definition of frontiers to take into account equivalence classes of the nodes as well:

Definition 11 (Class frontiers of a node) Given an egraph $G = \langle N, E, L, \text{root} \rangle$, the *class frontiers* of a node $n \in N$, denoted $c\mathcal{F}(n)$, is a set of sets of nodes defined as

$$\begin{aligned} c\mathcal{F}(n) = & \{\{\text{children}(n)\}\} \cup \\ & \{(\Pi \cup \{c\}) \setminus \{m\} \mid \Pi \in c\mathcal{F}(n) \wedge m \in \Pi \wedge c \in \text{class}(m)\} \cup \\ & \{(\Pi \cup \{\text{children}(m)\}) \setminus \{m\} \mid \Pi \in c\mathcal{F}(n) \wedge m \in \Pi \wedge \text{deg}(m) > 0\} \quad \square \end{aligned}$$

Lemma 12 *For any $n \in N$ s.t. $\text{deg}(n) > 0$, if $c\text{-ground}(n)$, there exists a class frontier $\Pi \in c\mathcal{F}(n)$ s.t. each node in Π is ground.* □

PROOF n is $c\text{-ground}$ if either (a) $\text{term}(n)$ is ground or (b) if $\text{deg}(n) > 0$ and for each $c \in \text{children}(n)$, there exists a $c\text{-ground}$ node in $\text{class}(c)$. To construct Π , we follow this definition. Initially, let $\Pi = \{\text{children}(n)\}$. For each c that have ground nodes in its class, we replace c with the ground node. For each non-ground, $c\text{-ground}$ node c' in Π , we replace c' with $\text{children}(c')$. We repeat this process until we get a class frontier with only ground nodes, the base case of the definition. ■

Lemma 13 *Let $r_2 = \text{process}(r_1, \text{todo})$. For any node $n \in N$ s.t. $\exists \Pi \in c\mathcal{F}(n) \cdot \Pi \subseteq \text{todo}$, $r_2(n) \neq \star$.* □

PROOF We prove it by induction on the class frontiers of n . The base case, when $\Pi = \{\text{children}(n)\}$, follows from Lem. 10. For the inductive step, assume that the lemma holds for a set $\Pi \in c\mathcal{F}(n)$. That is, if $\Pi \subseteq \text{todo}$ and $r_2 = \text{process}(r_1, \text{todo})$, $r_2(n)$ is defined. We prove that the lemma holds for both cases: (a) $\Pi' = (\Pi \cup \{\text{children}(m)\}) \setminus \{m\}$, where m is a non-leaf node in Π , and (b) $\Pi'' = (\Pi \cup \{m'\}) \setminus \{m\}$, where $m \in \Pi$ and $m' \in \text{class}(m)$. Case (a) follows from the same arguments used in the proof of Lem. 10. Case (b) holds because both m' and m have the same representative, as they are in the same class. ■

Lemma 14 $\forall n \in N \cdot c\text{-ground}(n) \Rightarrow c\text{-ground}(\text{repr}(n))$ □

PROOF For all ground nodes, all frontiers consists of only ground nodes. In particular, they have a frontier where all nodes are both ground and leaf. Alg. 11 calls **process** with all ground, leaf nodes in *todo* (line 5). Therefore, by Lem. 10, this call to **process** assigns representatives to all ground nodes in the egraph. At this point, all ground nodes either have representatives or are in *todo*. Therefore, by Lem. 12, each c-ground node in the egraph has a frontier in *todo*. Hence, by Lem. 13, **process** assigns representatives for all c-ground nodes. ■

Next, we show that the **repr** constructed by Alg. 11 is admissible. To do so, we extend the definition of admissibility of representative functions to partial representative functions as follows.

Definition 12 (Admissible partial representative functions) Given an egraph $G = \langle N, E, L, \text{root} \rangle$, we say that a function $\text{repr} : N \rightarrow N \cup \{\star\}$, is an *admissible partial representative function* for G if:

- (a) $\forall n \in N \cdot (\text{repr}(n) \neq \star) \Rightarrow (\forall n' \in \text{class}(n) \iff \text{repr}(n) = \text{repr}(n'))$,
- (b) the graph $\langle N, E_{\text{repr}} \rangle$ is acyclic where $E_{\text{repr}} = \{(n, \text{repr}(c)) \mid n \in N, c \in \text{children}(n), \text{repr}(c) \neq \star\}$, and
- (c) $\forall n, n' \in N \cdot n = \text{repr}(n') \Rightarrow \forall c \in \text{children}(n) \cdot \text{repr}(c) \neq \star$. □

The first two conditions state that **repr** is admissible for all nodes that it defines. The third condition states that, for all nodes in the range of **repr**, the representatives for their children are also defined. This allows building admissible partial representative functions based on already admissible partial representative functions.

Lemma 15 *Let r be an admissible partial representative function. Let n be a node s.t. $r(n) = \star$ and $\forall c \in \text{children}(n) \cdot r(c) \neq \star$. Let r_1 be a partial representative function defined as $r_1(n') = \text{ite}(n' \in \text{class}(n), n, r(n'))$. r_1 is an admissible partial representative function.* □

PROOF By condition (c) of Def. 12, for any child c of n and for any node m reachable from c in the graph $\langle N, E_r \rangle$, $r(m)$ is defined. Therefore, $r(m) \notin \text{class}(n)$ as $r(n)$ is undefined. Therefore, choosing n as representative of its equivalence class does not introduce any cycles. ■

Lemma 16 *Representative functions repr computed by find_defs are maximally ground and satisfy Def. 9.* □

PROOF Alg. 11 chooses a node n as class representative either if n is a leaf or if the representatives of all children of n have been chosen. Since choosing leaves as representatives does not introduce cycles, the first case preserves admissibility of partial representative functions. Lem. 15 shows that the second case also preserves admissibility of partial representative functions. By Lem. 11, repr is a total function. Ground maximality is stated in Lem. 14. ■

Finally, we prove that finding a maximally ground, admissible representative function is sufficient to find ground definitions for any variables that have it.

Theorem 8 *Let $G = \text{egraph}(\varphi)$ be an egraph and repr an admissible representative function that is maximally ground. For all $n \in N$, if $\varphi \models \text{term}(n) \approx t$ for some ground term t , then $\text{repr}(n)$ is c-ground and $\text{ntt}(\text{repr}(n))$ is ground.* □

PROOF From completeness of egraphs, it follows that if $\varphi \models \text{term}(n) \approx t$ for some ground term t , $\text{class}(n)$ is ground. By definition of maximally ground, for all such nodes, $\text{repr}(n)$ is c-ground. Next, we prove that for all representative nodes $r \in \text{range}(\text{repr})$ that are c-ground, $\text{ntt}(r)$ is ground.

We prove this by induction on G_{repr} . For the base case, all leaves $l \in \text{range}(\text{repr})$ s.t. $c\text{-ground}(l)$, $\text{ntt}(l)$ is ground. By definition of c-ground, leaves are c-ground if they are ground. Therefore, $\text{ntt}(l) = \text{term}(l)$ is ground. For the inductive case, we assume that, for a c-ground representative node n , all its children $n[i]$ have the property that $\text{ntt}(\text{repr}(n[i]))$ is ground. Therefore, for $f = L(n)$, and args s.t. $\text{args}[i] = \text{ntt}(\text{repr}(n[i]))$, $\text{ntt}(n) = f(\text{args})$ is also ground. ■

Theorem 8 implies that all terms that can be rewritten into ground equivalent ones will be rewritten, which, in turn, means that for each variable that has a ground definition, its representative is one such definition.

Finding Additional (Non-ground) Definitions. At this point, QEL found ground definitions while avoiding cycles in G_{repr} . However, this does not mean that as many variables as possible are eliminated. A variable can also be eliminated if it can

Algorithm 12: Refining `repr` and building core.

<pre> 1 function <i>egraph</i> :: refine_defs(<i>repr</i>, $\bar{v}$): 2 for $n \in N$ do 3 if $n = \text{repr}(n)$ and $L(n) \in \bar{v}$ then 4 $r := n$ 5 for $n' \in \text{class}(n) \setminus \{n\}$ do 6 if $L(n') \notin \bar{v}$ then 7 if not $\text{cycle}(n', \text{repr})$ then 8 $r := n'$; 9 break 10 for $n' \in \text{class}(n)$ do 11 $\text{repr}[n'] := r$ 12 return <i>repr</i> </pre>	<pre> 13 function <i>egraph</i> :: find_core(<i>repr</i>, $\bar{v}$): 14 <i>core</i> := $\emptyset$ 15 for $n \in N$ s.t. $n = \text{repr}(n)$ do 16 <i>core</i> := <i>core</i> $\cup \{n\}$ 17 for $n' \in (\text{class}(n) \setminus n)$ do 18 if $L(n') \in \bar{v}$ then 19 continue 20 else if $\exists m \in \text{core} \cdot m$ congruent with n' 21 then 22 continue 23 <i>core</i> := <i>core</i> $\cup \{n'\}$ 24 return <i>core</i> </pre>
--	--

be expressed as a function of other variables. This is not achieved by `find_defs`. For example, in `repr4.4b` both variables are representatives, hence none is eliminated, even though, since $x \approx g(f(y))$, x could be eliminated in φ_5 by rewriting x as a function of y . Alg. 12 shows function `refine_defs` that refines maximally ground `reprs` to further find such definitions while keeping admissibility and ground maximality. This is done by greedily attempting to change class representatives if they are labeled with a variable. `refine_defs` iterates over the nodes in the class checking if there is a different node that is not a variable and that does not create a cycle in G_{repr} (line 7). The resulting `repr` remains maximally ground because representatives of ground classes are not changed.

For example, let us refine `repr4.4b` = $\{\mathbf{N}(3), \mathbf{N}(6), \mathbf{N}(5)\}$ obtained for $\varphi_{4.4}$. Assume that x is processed first. For $\text{class}(\mathbf{N}(x))$, changing the representative to $\mathbf{N}(1)$ does not introduce a cycle (see Fig. 4.4c), so $\mathbf{N}(1)$ is selected. Next, for $\text{class}(\mathbf{N}(y))$, choosing $\mathbf{N}(4)$ causes G_{repr} to be cyclic since $\mathbf{N}(1)$ was already chosen (Fig. 4.4a), so the representative of $\text{class}(\mathbf{N}(y))$ is not changed. The final refinement is `repr4.4c` = $\{\mathbf{N}(1), \mathbf{N}(6), \mathbf{N}(5)\}$.

At this point, QEL found a representative function `repr` with as many ground definitions as possible and attempted to refine `repr` to have fewer variables as representatives. Next, QEL finds a core of the nodes of the egraph, based on `repr`, that will govern the translation of the egraph to a formula. While `repr` determines the semantic rewrites of terms that enable variable elimination, it is the use of the core in the translation that actually eliminates them.

Variable Elimination Based on a Core. A *core* of an egraph $G = \langle N, E, L, \text{root} \rangle$ and a representative function `repr`, is a subset of the nodes $N_c \subseteq N$ such that $\psi_c = G.\text{to_formula}(\text{repr}, N \setminus N_c)$ satisfies $\text{isFormula}(G, \psi_c)$.

Alg. 12 shows pseudocode for `find_core` that computes a core of an egraph for a given representative function. The idea is that non-representative nodes that are labeled by variables, as well as nodes congruent to nodes that are already in the core, need not be included in the core. The former are not needed since we are only interested in preserving the existential closure of the output, while the latter are not needed since congruent nodes introduce the same syntactic terms in the output. For example, for φ_1 and `repr1`, `find_core` returns `core1` = $N_1 \setminus \{N(3), N(5), N(9)\}$. Nodes $N(3)$ and $N(9)$ are excluded because they are labeled with variables; and node $N(5)$ because it is congruent with $N(4)$.

Finally, QEL produces a quantifier reduction by applying `to_formula` with the computed `repr` and `core`. Variables that are not in the core (they are not representatives) are eliminated – this includes variables that have a ground definition. However, QEL may eliminate a variable even if it is a representative (and thus it is in the core). As an example, consider $\psi(x, y) \triangleq f(x) \approx f(y) \wedge x \approx y$, whose egraph G contains 2 classes with 2 nodes each. The core N_c relative to any admissible `repr` contains only one representative per class: in the $class(N(x))$ because both nodes are labeled with variables, and in the $class(N(f(x)))$ because nodes are congruent. In this case, `to_formula(repr, Nc)` results in $\top$ (since singleton classes in the core produce no literals in the output formula), a quantifier elimination of ψ . More generally, the variables are eliminated because none of them is reachable in G_{repr} from a non-singleton class in the core (only such classes contribute literals to the output).

We conclude the presentation of QEL by showing its output for our examples. For φ_1 , QEL obtains $(k + 1 \approx read(a, x) \wedge 3 > k + 1)$, a quantifier reduction, using `repr1` = $\{N(3), N(8)\}$ and `core1` = $N_1 \setminus \{N(3), N(5), N(9)\}$. For $\varphi_{4.3}$, QEL obtains $(6 \approx f(g(6)))$, a quantifier elimination, using `repr4.3b` = $\{N(4), N(1)\}$, and `core4b` = $N_4 \setminus \{N(3), N(2)\}$. Finally, for $\varphi_{4.4}$, QEL obtains $(y \approx h(f(y)) \wedge f(g(f(y))) \approx f(y))$, a quantifier reduction, using `repr4.4c` = $\{N(1), N(6), N(5)\}$ and `core5c` = $N_5 \setminus \{N(3)\}$.

Guarantees of QEL. Correctness of QEL is straightforward. We conclude this section by providing two conditions that ensure that a variable is eliminated by QEL. The first condition guarantees that a variable is eliminated whenever a ground definition for it exists (regardless of the specific representative function and core computed by QEL). This makes QEL *complete relative to quantifier elimination based on ground definitions*. Relative completeness is an important property since it means that QEL is unaffected by variable orderings and syntactic rewrites, unlike QELITE. The second condition, illustrated by ψ above, depends on the specific representative function and core computed by QEL.

Theorem 9 *Let φ be a QF conjunction of literals with free variables $\bar{v}$, and let $v \in \bar{v}$. Let $G = \text{egraph}(\varphi)$, n_v the node in G such that $L(n_v) = v$ and `repr` and `core` computed*

by QEL. We denote by $NS = \{n \in \text{core} \mid (\text{class}(n) \cap \text{core}) \neq \{n\}\}$ the set of nodes from classes with two or more nodes in **core**. If one of the following conditions hold, then v does not appear in $QEL(\varphi, \bar{v})$:

- (1) there exists a ground term t s.t. $\varphi \models v \approx t$, or
- (2) n_v is not reachable from any node in NS in G_{repr} . □

PROOF For condition (1), variables that have a ground representative are excluded from **core**, therefore, they are excluded in **to_formula**, and are successfully eliminated.

For condition (2), **to_formula** calls **ntt** only on nodes in NS , this triggers a call to **ntt** only for all representatives of the children, i.e., only the ones reachable in G_{repr} . Therefore, if the node is not reachable in G_{repr} , it does not appear in the output formula. ■

As a corollary, if every variable meets one of the two conditions, then QEL finds a quantifier elimination.

This concludes the presentation of our quantifier reduction algorithm. Next, we show how QEL can be used to under-approximate quantifier elimination, which allows working with formulas for which QEL does not result in a qelim.

4.4 Model Based Projection Using QEL

Applications like model checking and quantified satisfiability require efficient computation of under-approximations of quantifier elimination. They make use of Model Based Projection (MBP, see Sec. 1.3) algorithms to project variables that cannot be eliminated cheaply. Our QEL algorithm is efficient and relatively complete, but it does not guarantee to eliminate all variables. In this section, we use a model and theory-specific projection rules to implement an MBP algorithm on top of QEL.

We focus on two important theories: Arrays and Algebraic DataTypes (ADT). They are widely used to encode program verification tasks. Prior works separately develop MBP algorithms for Arrays [70] and ADTs [15]. Both MBPs were presented as a set of syntactic rewrite rules applied until fixed point.

Combining the MBP algorithms for Arrays and ADTs is non-trivial because applying projection rules for one theory may produce terms of the other theory. Therefore, separately achieving saturation in either theory is not sufficient to reach saturation in the combined setting. The MBP for the combined setting has to call both MBPs, check whether either one of them produced terms that can be processed

$$\frac{\text{ELIMWRD1} \quad \varphi[\text{read}(\text{write}(t, i, v), j)]}{\varphi[v] \wedge i \approx j} M \models i \approx j$$

$$\frac{\text{ELIMWRD2} \quad \varphi[\text{read}(\text{write}(t, i, v), j)]}{\varphi[\text{read}(t, j)] \wedge i \not\approx j} M \models i \not\approx j$$

$$\frac{\text{PartialEq} \quad \varphi[e_1 \approx e_2]}{\varphi[e_1 =_0^p e_2]}$$

$$\frac{\text{ElimEq} \quad \exists v \cdot v =_i^p e \wedge \varphi}{\varphi[v \rightarrow \text{write}(e, \bar{i}, \bar{d})]} \neg \text{has_var}(e, v)$$

$$\frac{\text{ACKERMANN1} \quad \begin{array}{l} \varphi \wedge \text{read}(v, t_1) \approx s_1 \wedge \\ \text{read}(v, t_2) \approx s_2 \end{array}}{\varphi \wedge s_1 \approx s_2 \wedge t_1 \approx t_2} M \models t_1 \approx t_2$$

$$\frac{\text{ACKERMANN2} \quad \begin{array}{l} \varphi \wedge \text{read}(v, t_1) \approx s_1 \wedge \\ \text{read}(v, t_2) \approx s_2 \end{array}}{\varphi \wedge t_1 \not\approx t_2} M \models t_1 \not\approx t_2$$

ElimWrRd
function *match* (*t*):
 return *t* = *read*(*write*(*s*, *i*, *v*), *j*)
function *apply* (*t*, *M*, *G*):
 if *M* $\models i \approx j$ **then**
 G.assert(*i* $\approx j$)
 G.assert(*t* $\approx v$)
 else
 G.assert(*i* $\not\approx j$)
 G.assert(*t* $\approx \text{read}(s, j)$)

PartialEq
function *match* (*t*):
 return *t* = *e*₁ $\approx$ *e*₂ $\wedge$ *has_vars*(*t*)
function *apply* (*t*, *M*, *G*):
 G.assert(*e*₁ =₀^{*p*} *e*₂)

ElimEq
function *match* (*t*):
 return *t* = *v* =_i^{*p*} *e* $\wedge$ *is_var*(*v*) $\wedge$ $\neg \text{has_var}(e, v)$
function *apply* (*t*, *M*, *G*):
 $\bar{d} := \text{newVars}()$
 G.assert(*v* $\approx \text{write}(e, \bar{i}, \bar{d})$)
 G.assert(*t* $\approx \top$)
 $\forall 0 \leq j < \text{len}(\bar{i}) \cdot M[\bar{d}_j] := M[\text{select}(v, \bar{i}_j)]$

Ackermann
function *match* ($\langle t_1, t_2 \rangle$):
 return *t*₁ = *read*(*v*, *e*₁) $\wedge$
 $t_2 = \text{read}(v, e_2) \wedge e_1 \neq e_2$
function *apply* (*t*₁, *t*₂, *M*, *G*):
 if *M* $\models e_1 \approx e_2$ **then**
 G.assert(*e*₂ $\approx e_2$)
 else
 G.assert(*e*₁ $\not\approx e_2$)

Figure 4.5: MBP rules from [70]. The notation $\varphi[t]$ means that φ contains term t . For each MBP rule on the left hand side, there is a corresponding egraph implementation. The **match** method pattern matches input terms and the **apply** method assumes that the input terms satisfy the match method and uses patterns from the match to do rewriting.

by the other, and, if so, call the other algorithm. This is similar to theory combination in SMT solving where the core SMT solver has to keep track of different theory solvers and exchange terms between them.

Our main insight is that egraphs can be used as a glue to combine MBP algorithms for different theories, just like egraphs are used in SMT solvers to combine satisfiability checking for different theories. Implementing MBP using egraphs allows us to use the insights from QEL to combine MBP with on-the-fly quantifier reduction to produce less under-approximate formulas than what we get by syntactic application of MBP rules.

To implement MBP using egraphs, we implement all rewrite rules for MBP in Arrays [70] and ADTs [15] on top of egraphs. We explain the implementation of the MBP rules for Arrays here.

Fig. 4.5 shows Array MBP rules from [70] along with their egraph implementations. Consider the ELIMWRD1 and ELIMWRD2 rules. Here, φ is a formula with arrays and M is a model for φ . Both rules rewrite terms which match the pattern $\text{read}(\text{write}(t, i, v), j)$, where t, i, j, k are all terms and t contains a variable to be projected. ELIMWRD1 is applicable when $M \models i \approx j$. It rewrites the term $\text{read}(\text{write}(t, i, v), j)$ to v . ELIMWRD2 is applicable when $M \not\models i \approx j$ and rewrites $\text{read}(\text{write}(t, i, v), j)$ to $\text{read}(t, j)$.

ElimWrRd in Fig. 4.5 shows the adaptation of ELIMWRD1 and ELIMWRD2 rules to egraph. The `match(t)` method checks if t syntactically matches $\text{read}(\text{write}(s, i, v), j)$, where s contains a variable to be projected. The `apply(t)` method assumes that t is $\text{read}(\text{write}(s, i, v), j)$. It first checks if $M \models i \approx j$, and, if so, it adds $i \approx j$ and $t \approx v$ to the egraph G . Otherwise, if $M \not\models i \approx j$, `apply(t)` adds a disequality $i \not\approx j$ and an equality $t \approx \text{read}(s, v)$ to G . That is, the egraph implementation of the rules only adds (and does not remove) literals that capture the side condition and the conclusion of the rule.

Our algorithm for MBP based on egraphs, MBP-QEL, is shown in Alg. 13. It initializes an egraph with the input formula (line 2), applies MBP rules until saturation (line 5), and then uses the steps of QEL (lines 8–13) to generate the projected formula.

Applying rules is as straightforward as iterating over all terms t in the egraph, and for each rule r such that $r.\text{match}(t)$ is true, calling $r.\text{apply}(t, M, G)$ (lines 16–25). As opposed to the standard approach based on formula rewriting, here the terms are *not* rewritten – both remain. Therefore, it is possible to get into an infinite loop by re-applying the same rules on the same terms over and over again. To avoid this, MBP-QEL marks terms as *seen* (line 26) and avoids them in the next iteration (line 17). Some rules in MBP are applied to pairs of terms. For example, ACKERMANN rewrites pairs of *read* terms over the same variable. This is different from usual applications

where rewrite rules are applied to individual expressions. Yet, it is easy to adapt such pairwise rewrite rules to egraphs by iterating over pairs of terms (lines 28–33).

MBP-QEL does not apply MBP rules to terms that contain variables but are already c-ground (line 18), which is sound because such terms are replaced by ground terms in the output (theorem 9). This prevents unnecessary application of MBP rules thus allowing MBP-QEL to compute MBPs that are closer to a quantifier elimination (less model-specific).

Just like each application of a rewrite rule introduces a new term to a formula, each call to the `apply` method of a rule adds new terms to the egraph. Therefore, each call to `ApplyRules` (line 5) makes the egraph bigger. However, provided that the original MBP combination is terminating, the iterative application of `ApplyRules` terminates as well (due to marking).

Some MBP rules introduce new variables to the formula. MBP-QEL computes `repr` based on both original and newly introduced variables (line 8). This allows MBP-QEL to eliminate all variables, including non-Array, non-ADT variables, that are equivalent to ground terms (theorem 9).

As mentioned earlier, MBP-QEL never removes terms while rewrite rules are saturating. Therefore, after saturation, the egraph still contains all original terms and variables. From soundness of the MBP rules, it follows that after each invocation of `apply`, MBP-QEL creates an under-approximation of $\varphi^\exists$ based on the model M . From completeness of MBP rules, it follows that, after saturation, all terms containing Array or ADT variables can be removed from the egraph without affecting equivalence of the saturated egraph. Hence, when calling `to_formula`, MBP-QEL removes all terms containing Array or ADT variables (line 13). This includes, in particular, all the terms on which rewrite rules were applied, but potentially more. We demonstrate our MBP algorithm on an example with nested ADTs and Arrays. Let $P \triangleq \langle A_{I \times I}, I \rangle$ be the datatype of a pair of an integer array and an integer, and let $pair : A_{I \times I} \times I \rightarrow P$ be its sole constructor with destructors $fst : P \rightarrow A_{I \times I}$ and $snd : P \rightarrow I$. In the following, let i, l, j be integers, a an integer array, p, p' pairs, and $\bar{p}_1, \bar{p}_2$ arrays of pairs ($A_{I \times P}$). Consider the formula:

$$\varphi_{mbp}(p, a) \triangleq read(a, i) \approx i \wedge p \approx pair(a, l) \wedge \bar{p}_2 \approx write(\bar{p}_1, j, p) \wedge p \not\approx p'$$

where p and a are free variables that we want to project and all of $i, j, l, \bar{p}_1, \bar{p}_2, p'$ are constants that we want to keep. MBP is guided by a model $M_{mbp} \models \varphi_{mbp}$. To eliminate p and a , MBP-QEL constructs the egraph of φ_{mbp} and applies the MBP rules. In particular, it uses Array MBP rules to rewrite the $write(\bar{p}_1, j, p)$ term by adding the equality $read(\bar{p}_2, j) \approx p$ and merging it with the equivalence class of $\bar{p}_2 \approx write(\bar{p}_1, j, p)$. It then applies ADT MBP rules to deconstruct the equality $p \approx pair(a, l)$ by creating two equalities $fst(p) \approx a$ and $snd(p) \approx l$. Finally, the call to

Algorithm 13: MBP-QEL: an MBP using QEL. Here $gr(t, \bar{v})$ checks whether term t contains any variables in $\bar{v}$ and $is_eq(t)$ checks if t is an equality literal.

In: A QF formula φ with free variables $\bar{v}$ all of sort $Array(I, V)$ or ADT, a model $M \models \varphi^\exists$, and sets of rules $ArrayRules$ and $ADTRules$ Out: A cube ψ s.t. $\psi^\exists \Rightarrow \varphi^\exists$, $M \models \psi^\exists$, and $vars(\psi)$ are not Arrays or ADTs <pre> 1 function MBP-QEL($\bar{v}, \varphi, M$): 2 $G := egraph(\varphi)$ 3 $p_1, p_2 := \top, \top$; $S, S_p := \emptyset, \emptyset$ 4 while $p_1 \vee p_2$ do 5 $p_1 := ApplyRules(G, M, ArrayRules, S, S_p)$ 6 $p_2 := ApplyRules(G, M, ADTRules, S, S_p)$ 7 $\bar{v}' := G.Vars()$ 8 $repr := G.find_defs(\bar{v}')$ 9 $repr := G.refine_defs(repr, \bar{v}')$ 10 $core := G.find_core(repr, \bar{v}')$ 11 $\bar{v}_e := \{v \in \bar{v}' \mid is_arr(v) \vee is_adt(v)\}$ 12 $core_e := \{n \in core \mid gr(term(n), \bar{v}_e)\}$ 13 return $G.to_formula(repr, G.Nodes() \setminus core_e)$ </pre>	<pre> 14 function ApplyRules(G, M, R, S, S_p): 15 $progress := \perp$ 16 $N := G.Nodes()$ 17 $U := \{n \mid n \in N \setminus S\}$ 18 $T := \{term(n) \mid n \in U \wedge$ 19 $(is_eq(term(n)) \vee \neg c_ground(n))\}$ 20 $R_p := \{r \in R \mid r.is_for_pairs()\}$ 21 $R_u := R \setminus R_p$ 22 foreach $t \in T, r \in R_u$ do 23 if $r.match(t)$ then 24 $r.apply(t, M, G)$ 25 $progress := \top$ 26 $S := S \cup N$ 27 $N_p := \{\langle n_1, n_2 \rangle \mid n_1, n_2 \in N\}$ 28 $T_p := \{term(n_p) \mid n_p \in N_p \setminus S_p\}$ 29 foreach $t_p \in T_p, r \in R_p$ do 30 if $r.match(p)$ then 31 $r.apply(p, M, G)$ 32 $progress := \top$ 33 $S_p := S_p \cup N_p$ 34 return $progress$ </pre>
--	---

to formula produces

$$\begin{aligned}
read(fst(read(\bar{p}_1, j)), i) &\approx i \wedge snd(read(\bar{p}_1, j)) \approx l \wedge \\
read(\bar{p}_2, j) &\approx pair(fst(read(\bar{p}_1, j)), l) \wedge \\
\bar{p}_2 &\approx write(\bar{p}_1, j, read(\bar{p}_2, j)) \wedge read(\bar{p}_2, j) \not\approx p'
\end{aligned}$$

The output is easy to understand by tracing it back to the input. For example, the first literal is a rewrite of the literal $read(a, i) \approx i$ where a is represented with $fst(p)$ and p is represented with $read(\bar{p}_1, j)$. While the interaction of these rules might seem straightforward in this example, the MBP implementation in Z3 fails to project a in this example because of the multilevel nesting.

Notably, in this example, the c-ground computation during projection allows MBP-QEL not splitting on the disequality $p \not\approx p'$ based on the model. While ADT MBP rules eliminate disequalities by using the model to split them, MBP-QEL benefits from the fact that, after the application of Array MBP rules, the class of p becomes ground, making $p \not\approx p'$ c-ground. Thus, the c-ground computation allows MBP-QEL to produce a formula that is less approximate than those produced by syntactic application of MBP rules. In fact, in this example, a quantifier elimination

is obtained (the model M_{mbp} was not used).

In the next section, we show that our improvements to MBP translate to significant improvements in SPACER.

4.5 Evaluation

We implement QEL (Alg. 10) and MBP-QEL (Alg. 13) inside Z3 version 4.12.0. Our implementation (referred to as Z3EG) has been integrated into Z3⁴. Z3EG replaces QELITE with QEL, and the existing MBP with MBP-QEL.

We evaluate Z3EG using two solving tasks. Our first evaluation is on the QSAT algorithm [15] for checking satisfiability of formulas with alternating quantifiers. In QSAT, Z3 uses both QELITE and MBP to under-approximate quantified formulas. We compare three QSAT implementations: the existing version in Z3 with the default QELITE and MBP; the existing version in Z3 in which QELITE and MBP are replaced by our egraph-based algorithms, Z3EG; and the QSAT implementation in YICESQS⁵, based on the YICES [31] SMT solver. During the evaluation, we found a bug in QSAT implementation of Z3 and fixed it⁶. The fix resulted in Z3 solving over 40 sat instances and over 120 unsat instances more than before. In the following, we use the fixed version of Z3.

We use benchmarks in the theory of (quantified) LIA and LRA from SMT-LIB [7], [8], with alternating quantifiers. LIA and LRA are the only tracks in which Z3 uses the QSAT tactic by default. To make our experiments more comprehensive, we also consider two modified variants of the LIA and LRA benchmarks, where we add some non-recursive ADT variables to the benchmarks. Specifically, we wrap all existentially quantified arithmetic variables using a record type ADT and unwrap them whenever they get used⁷. Since these benchmarks are similar to the original, we force Z3 to use the QSAT tactic on them with a `tactic.default_tactic=qsat` command line option.

Tab. 4.1 summarizes the results for the SMT-LIB benchmarks. In LIA, both Z3EG and Z3 solve all benchmarks in under a minute, while YICESQS is unable to solve many instances. In LRA, YICESQS solves all instances with very good performance. Z3 is able to solve only some benchmarks, and our Z3EG performs similarly to Z3. We found that in the LRA benchmarks, the new algorithms in Z3EG are not being

⁴It has been integrated into Z3 version 4.12.3 <https://github.com/Z3Prover/z3/releases/tag/z3-4.12.3>. See pull request <https://github.com/Z3Prover/z3/pull/6820> for more details

⁵Available at <https://github.com/disteph/yicesQS>.

⁶Available at <https://github.com/igcontreras/z3/commit/133c9e438ce>.

⁷The modified benchmarks are available at <https://github.com/igcontreras/LIA-ADT> and <https://github.com/igcontreras/LRA-ADT>.

Cat.	Count	Z3EG		Z3		YICESQS	
		SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
LIA	416	150	266	150	266	107	102
LRA	2 419	795	1 589	793	1 595	808	1 610

Table 4.1: Instances solved within 20 minutes by different implementations. Benchmarks are quantified **LIA** and **LRA** formulas from SMT-LIB [7].

Cat.	Count	Z3EG		Z3	
		SAT	UNSAT	SAT	UNSAT
LIA-ADT	416	150	266	150	56
LRA-ADT	2 419	757	1 415	196	964

Table 4.2: Instances solved within 60 seconds for our handcrafted benchmarks.

used since there are not many equalities in the formula, and no equalities are inferred during the run of QSAT. Thus, any differences between Z3 and Z3EG are due to inherent randomness of the solving process.

Tab. 4.2 summarizes the results for the categories of mixed ADT and arithmetic. YICESQS is not able to compete because it does not support ADTs. As expected, Z3EG solves many more instances than Z3.

The second part of our evaluation shows the efficacy of MBP-QEL for Arrays and ADTs (Alg. 13) in the context of SPACER. We compare Z3 and Z3EG on CHC problems containing Arrays and ADTs. We use two sets of benchmarks to test out the efficacy of our MBP. The benchmarks in the first set were generated for verification of Solidity smart contracts [3] (we exclude benchmarks with non-linear arithmetic, they are not supported by SPACER). These benchmarks have a very complex structure that nests ADTs and Arrays. Specifically, they contain both ADTs of Arrays, as well as Arrays of ADTs. This makes them suitable to test our MBP-QEL. Row 1 of Tab. 4.3 shows the number of instances solved by Z3 (SPACER) with and without MBP-QEL. Z3EG solves 29 instances more than Z3. Even though MBP is just one part of the overall SPACER algorithm, we see that for these benchmarks, MBP-QEL makes a significant impact on SPACER. Digging deeper, we find that many of these instances come from the category called *abi* (row 2 in Tab. 4.3). Z3EG solves all of these benchmarks, while Z3 fails to solve 20 of them. We traced the problem down to the MBP implementation in Z3: it fails to eliminate all variables, causing runtime exception. In contrast, MBP-QEL eliminates all variables successfully, allowing Z3EG to solve these benchmarks.

We also compare Z3EG with ELDARICA [61], a state-of-the-art CHC-solver that is particularly effective on these benchmarks. Z3EG solves almost as many instances as ELDARICA. Furthermore, like Z3, Z3EG is orders of magnitude faster than ELDARICA. Finally, we compare the performance of Z3EG on Array benchmarks from the CHC

Cat.	Count	Z3EG		Z3		ELDARICA	
		SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
Solidity	3 468	2 324	1 133	2 314	1 114	2 329	1 134
↳ abi	127	19	108	19	88	19	108
LIA-lin-Arrays	488	214	72	212	75	147	68

Table 4.3: Instances solved within 20 minutes by Z3EG, Z3, and ELDARICA. Benchmarks are CHCs from **Solidity** [3] and CHC competition [56]. The **abi** benchmarks are a subset of **Solidity** benchmarks.

competition [56]. Z3EG is competitive with Z3, solving 2 additional safe instances and almost as many unsafe instances as Z3 (row 3 of Tab. 4.3). Both Z3EG and Z3 solve quite a few instances more than ELDARICA.

Our experiments show the effectiveness of our QEL and MBP-QEL in different settings inside the state-of-the-art SMT solver Z3. While we maintain performance on quantified arithmetic benchmarks, we improve Z3’s QSAT algorithm on quantified benchmarks with ADTs. On verification tasks, QEL and MBP-QEL help SPACER solve 30 new instances, even though MBP is only a relatively small part of the overall SPACER algorithm.

4.6 Conclusion

Quantifier elimination, and its under-approximation, Model-Based Projection are used by many SMT-based decision procedures, including quantified SAT and Constrained Horn Clause solving. Traditionally, these are implemented by a series of syntactic rules, operating directly on the syntax of an input formula. We argue that these procedures should be implemented directly on the egraph data-structure, already used by most SMT solvers. This results in algorithms that better handle implicit equality reasoning and result in easier to implement and faster procedures. We justify this argument by implementing quantifier reduction and MBP in Z3 using egraphs and show that the new implementation translates into significant improvements to the target decision procedures. Thus, our work provides both theoretical foundations for quantifier reduction and practical contributions to Z3 SMT-solver.

Chapter 5

Speculative SAT Modulo SAT

IC3/PDR is an efficient SAT-based Model Checking algorithm. Among many other innovations in IC3/PDR is the concept of a modular SAT-solver that divides a formula into multiple *frames* and each frame is solved by an individual SAT solver. The solvers communicate by exchanging proof obligations (i.e., satisfying assignments) and lemmas (i.e., learned clauses).

While modular reasoning in IC3/PDR is very efficient for a Model Checker, it is not as efficient as a classical monolithic SAT-solver. This is not surprising since modularity restricts the solver to colorable refutations [58], which are, in the worst case, exponentially bigger than unrestricted refutations. On the positive side, IC3/PDR’s modular SAT-solving makes interpolation trivial, and enables generalizations of proof obligations and inductive generalization of lemmas – both are key to the success of IC3/PDR.

This motivates the study of modular SAT-solving, initiated by SMS [9]. Our strategic vision is that our study will contribute to improvements in IC3/PDR. However, in this chapter, we focus on modular SAT-solving in isolation.

In modular SAT-solving, multiple solvers interact to check satisfiability of a partitioned CNF formula, where each part of the formula is solved by one of the solvers. For simplicity, we consider the case of two solvers $\langle S_s, S_m \rangle$ checking satisfiability of a formula pair $\langle \Phi_s, \Phi_m \rangle$. S_m is a *main* solver and S_s is a *secondary* solver. In the notation, the solvers are written right-to-left to align with IC3/PDR, where the main solver is used for frame 1 and the secondary solver is used for frame 0.

When viewed as a modular SAT-solver, IC3/PDR is uni-directional. First, S_m finds a satisfying assignment σ to Φ_m and only then, S_s extends σ to an assignment for Φ_s . Learned clauses, called *lemmas* in IC3/PDR, are only shared (or copied) from the secondary solver S_s to the main solver S_m .

SAT Modulo SAT (SMS) [9] is a modular SAT-solver that extends IC3/PDR by

allowing inter-modular unit propagation and conflict analysis: whenever an interface literal is placed on a trail of any solver, it is shared with the other solver and both solvers run unit propagation, exchanging unit literals. This makes modular SAT-solving in SMS bi-directional as information flows in both directions between the solvers. Bi-directional reasoning can simplify proofs, but it significantly complicates conflict analysis. To manage conflict analysis, SMS does not allow the secondary solver S_s to make any decisions before the main solver S_m is able to find a complete assignment to its clauses. As a result, learned clauses are either local to each solver, or flow only from S_s to S_m , restricting the structure of refutations similarly to IC3/PDR.

Both IC3/PDR and SMS require S_m to find a complete satisfying assignment to Φ_m before the solving is continued in S_s . This is problematic since Φ_m might be hard to satisfy, causing them to get stuck in Φ_m , even if considering both formulas together quickly reveals the (un)satisfiability of $\langle \Phi_s, \Phi_m \rangle$.

We introduce SPECSMS — a modular SAT-solver that employs a truly bi-directional reasoning. SPECSMS builds on SMS, while facilitating deeper communication between the modules by (1) allowing learnt clauses to flow in both directions, and (2) letting the two solvers interleave their decisions. The key challenge is in the adaptation of conflict analysis to properly handle the case of a conflict that depends on decisions over local variables of both solvers. Such a conflict cannot be explained to either one of the solvers using only interface clauses (i.e., clauses over interface variables). It may, therefore, require backtracking the search without learning any conflict clauses. To address this challenge, SPECSMS uses *speculation*, which tames decisions of the secondary solver that are interleaved with decisions of the main solver. If the secondary solver satisfies all of its clauses during speculation, a *validation* phase is employed, where the main solver attempts to extend the assignment to satisfy its unassigned clauses. If speculation leads to a conflict which depends on local decisions of both solvers, *refinement* is employed to resolve the conflict. Refinement ensures progress even if no conflict clause can be learnt. With these ingredients, we show that SPECSMS is sound and complete (i.e., always terminates).

To certify SPECSMS’s result when it determines that a formula is unsatisfiable, we extract a *modular* clausal proof from its execution. To this end, we extend DRUP proofs [59] to account for modular reasoning, and devise a procedure for trimming modular proofs. Such proofs are applicable both to SPECSMS and to SMS. Finally, we propose an interpolation algorithm that extracts an interpolant [26] from a modular proof. Since clauses are propagated between the solvers in both directions, the extracted interpolants have the shape $\bigwedge_i (C_i \Rightarrow cls_i)$, where C_i are conjunctions of clauses and each cls_i is a clause.

Original SMS is implemented on top of MiniSAT. We implemented both SMS and SPECSMS in Z3, using the extendable SAT-solver interface of Z3. Thanks to

its bi-directional reasoning, SPECSMS is able to efficiently solve both sat and unsat formulas that are provably hard for existing modular SAT-solvers, provided that speculation is performed at the right time. We describe a simple heuristic to decide when to speculate.

In summary, we make the following contributions: (i) the SPECSMS algorithm that leverages bi-directional modular reasoning (Sec. 5.2); (ii) modular DRUP proofs for SPECSMS (Sec. 5.3.1); (iii) proof-based interpolation algorithm; (iv) heuristics to guide speculation (Sec. 5.4); and (v) implementation and validation (Sec. 5.5).

This chapter has been published in the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024) [66].

5.1 Motivating examples

In this section, we discuss two examples in which both IC3/PDR-style uni-directional reasoning and SMS-style shallow bi-directional reasoning are ineffective. The examples illustrate why existing modular reasoning gets stuck. To better convey our intuition, we present our problems at word level using bit-vector variables directly, without explicitly converting them to propositional variables.

Example 11 Consider the following modular sat query: $\langle \varphi_{in}, \varphi_{\text{SHA-1}} \rangle$, where $\varphi_{in} \triangleq (in = in_1) \vee (in = in_2)$, in is a 512-bit vector, in_1, in_2 are 512-bit values, $\varphi_{\text{SHA-1}} \triangleq (\text{SHA-1}_{\text{circ}}(in) = \text{SHA-1}_{in_1})$, $\text{SHA-1}_{\text{circ}}(in)$ is a circuit that computes SHA-1 of in , and SHA-1_{in_1} is the 20 byte SHA-1 message digest of in_1 .

Checking the satisfiability of $\varphi_{in} \wedge \varphi_{\text{SHA-1}}$ is easy because it contains both the output and the input of the SHA-1 circuit. However, existing modular SAT-solvers attempt to solve the problem starting by finding a complete satisfying assignment to $\varphi_{\text{SHA-1}}$. This is essentially the problem of inverting the SHA-1 function, which is known to be very hard for a SAT-solver. The improvements in SMS allow unit propagation between the two modules. However, this does not help since there are no unit clauses in φ_{in} .

On the other hand, SPECSMS proceeds as follows: (1) when checking satisfiability of $\varphi_{\text{SHA-1}}$, it decides to speculate, (2) it starts checking satisfiability of φ_{in} , branches on variables in , finds an assignment σ to in and unit propagates σ to $\varphi_{\text{SHA-1}}$, (3) if there is a conflict in $\varphi_{\text{SHA-1}}$, it learns the conflict clause $in \neq in_2$, and (4) it terminates with a satisfying assignment $in = in_1$. Speculation in step (1) is what differentiates SPECSMS from IC3/PDR and SMS. The specifics of when exactly SPECSMS speculates is guided by a heuristic that is explained in Sec. 5.4. $\square$

Example 12 Speculation is desirable for unsatisfiable formulas as well. Consider the modular sat query $\langle \varphi_+, \varphi_- \rangle$, where $\varphi_+ \triangleq (a < 0 \Rightarrow x) \wedge (a \geq 0 \Rightarrow x) \wedge PHP_{32}^1$ and $\varphi_- \triangleq (b < 0 \Rightarrow \neg x) \wedge (b \geq 0 \Rightarrow \neg x) \wedge PHP_{32}^2$. Here, a and b are 32-wide bitvectors and local to the respective modules. PHP_{32} encodes the problem of fitting 32 pigeons into 31 holes and PHP_{32}^1 and PHP_{32}^2 denote a partitioning of PHP_{32} into 2 problems such that both formulas contain all variables. The modular problem $\langle \varphi_+, \varphi_- \rangle$ is unsatisfiable, x and PHP_{32}^1 being two possible interpolants. IC3/PDR and SMS only find the second interpolant. This is because, all satisfying assignments to φ_- immediately produce a conflict in PHP_{32}^1 part of φ_+ , without having to make any decisions. However, learning an interpolant containing x requires searching (i.e., deciding) in both φ_+ and φ_- . SPECSMS solves this problem by speculating right after deciding on all b variables. During speculation, the secondary solver hits a conflict on x once it tries to find an assignment to a variables. Note here that speculating after finding assignments to b variables and before finding an assignment to PHP_{32}^2 is crucial for SPECSMS to find the small interpolant. $\square$

These examples highlight the need to speculate while doing modular reasoning. Even though speculation by itself is quite powerful, to make SPECSMS effective in practice, we need good heuristics to decide when to enter speculation. We discuss some simple heuristics in Sec. 5.4.

5.2 Speculative SAT Modulo SAT

This section presents SPECSMS — a modular bi-directional SAT algorithm. For simplicity, we restrict our attention to the case of two modules. However, the algorithm easily generalizes to any sequence of modules.

5.2.1 Sat Modulo Sat

We assume that the reader has some familiarity with internals of a MiniSAT-like SAT solver [34] and with SMS [9]. We give a brief background on SMS, highlighting some of the key aspects. SMS decides satisfiability of a partitioned CNF formula $\langle \Phi_s, \Phi_m \rangle$ with a set of shared interface variables I . It uses two modules $\langle S_s, S_m \rangle$, where S_m is a *main* module used to solve Φ_m , and S_s is a *secondary* module to solve Φ_s . Each module is a SAT solver (with a slightly extended interface, as described in this section). We refer to them as *modules* or *solvers*, interchangeably. Each solver has its own clause database (initialized with Φ_i for $i \in \{m, s\}$), and a trail of literals, just as a regular SAT solver. The solvers keep their decision levels in sync. Whenever a decision is

made in one solver, the decision level of the other solver is incremented as well (adding a *null* literal to its trail if necessary). Whenever one solver back-jumps to level i , the other solver back-jumps to level i as well. Assignments to interface variables are shared between the solvers: whenever such a literal is added to the trail of one solver (either as a decision or due to propagation), it is also added to the trail of the other solver. SMS requires that S_s does not make any decisions, until S_m finds a satisfying assignment to its clauses.

Inter-modular propagation and conflict analysis The two key features of SMS are inter-modular unit propagation (called `PROPAGATEALL` in [9]) and the corresponding inter-modular conflict analysis. In `PROPAGATEALL`, whenever an interface literal is added to the trail of one solver, it is added to the trail of the other, and both solvers run unit propagation. Whenever a unit literal ℓ is copied from the trail of one solver to the other, the **reason** for ℓ in the destination solver is marked using a marker `ext`. This indicates that the justification for the unit is external to the destination solver¹. Propagation continues until either there are no more units to propagate or one of the solvers hits a conflict.

Conflict analysis in SMS is extended to account for units with no reason clauses. If such a literal ℓ is used in conflict analysis, its reason is obtained by using `AnalyzeFinal( $\ell$ )` on the other solver to compute a clause ($s \Rightarrow \ell$) over the interface literals. This clause is copied to the requesting solver and is used as the missing reason. Multiple such clauses can be copied (or learned) during analysis of a single conflict clause – one clause for each literal in the conflict that is assigned by the other solver.

In SMS, it is crucial that `AnalyzeFinal( $\ell$ )` always succeeds to generate a reason clause over the interface variables. This is ensured by only calling `AnalyzeFinal( $\ell$ )` in the S_s solver on literals that were added to the trail when S_s was not yet making decisions. This can happen in one of two scenarios: either S_m hits a conflict due to literals propagated from S_s , in which case `AnalyzeFinal` is invoked in S_s on each literal marked `ext` in S_m that is involved in the conflict resolution to obtain its **reason**; or S_s hits a conflict during unit propagation, in which case it invokes `AnalyzeFinal` to obtain a conflict clause over the interface variables that blocks the partial assignment of S_m . In both cases, new reason clauses are always copied from S_s to S_m . We refer the reader to [9] for the pseudo-code of the above inter-modular procedures for details.

¹This is similar to theory propagation in SMT solvers.

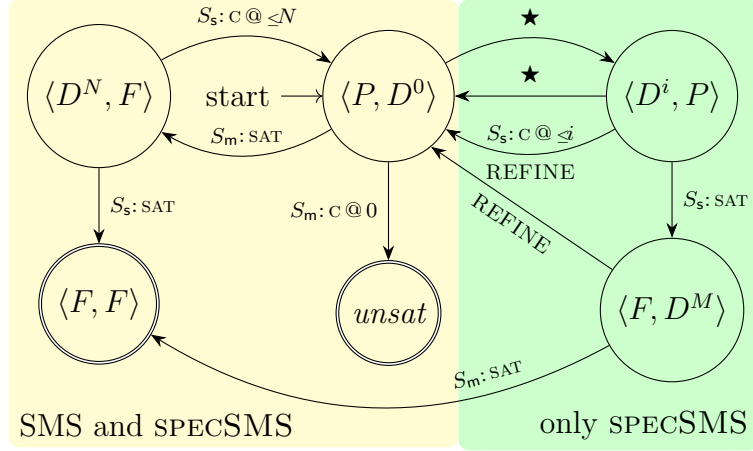


Figure 5.1: State transitions of SPECSMS. A state $\langle P, D^0 \rangle$ means that the secondary solver S_s is in propagate mode and the main solver S_m is in decide mode. Each edge is guarded with a condition. The condition $S_m: SAT$ means that S_m found a full satisfying assignment to Φ_m . The condition $S_m: C @ \leq j$ means that S_m hit a conflict at a decision level below j . The four states in yellow corresponds to SMS; two states in green are unique to SPECSMS.

5.2.2 Speculative Sat Modulo Sat

SPECSMS extends SMS [9] by a combination of *speculation*, *refinement*, and *validation*. During the search in the main solver S_m , SPECSMS non-deterministically *speculates* by allowing the secondary solver S_s to extend the current partial assignment of Φ_m to a satisfying assignment of Φ_s . If S_s is unsuccessful (i.e., hits a conflict), and the conflict depends on a combination of a *local* decision of S_m with some decision of S_s , then the search reverts to S_m and its partial assignment is *refined* by forcing S_m to decide on an interface literal from the conflict. On the other hand, if S_s is successful, solving switches to the main solver S_m that *validates* the current partial assignment by extending it to all of its clauses. This either succeeds (meaning, $\langle \Phi_s, \Phi_m \rangle$ is sat), or fails and another *refinement* is initiated. Note that the two sub-cases where S_s is unsuccessful but the reason for the conflict is either local to S_s or local to S_m are handled as in SMS.

Search modes SPECSMS controls the behavior of the solvers and their interaction through *search modes*. Each solver can be in one of the following search modes: Decide, Propagate, and Finished. In Decide, written D^i , the solver treats all decisions below level i as assumptions and is allowed to both make decisions and do unit propagation.

In Propagate, written P , the solver makes no decisions, but does unit propagation whenever new literals are added to its trail. In Finished, written F , the clause database of the solver is satisfied; the solver neither makes decisions nor propagates unit literals.

The pair of search modes of both modules is called the *state* of SPECSMS, where we add a unique state called *unsat* for the case when the combination of the modules is known to be unsatisfiable. The possible states and transitions of SPECSMS are shown in Fig. 5.1. States *unsat* and $\langle F, F \rangle$ are two final states, corresponding to unsat and sat, respectively. In all other states, exactly one of the solvers is in a state D^i . We refer to this solver as *active*. The part of the transition system highlighted in yellow correspond to SMS, and the green part includes the states and transitions that are unique to SPECSMS.

Normal execution with bi-directional propagation SPECSMS starts in the state $\langle P, D^0 \rangle$, with the main solver being active. In this state, it can proceed like SMS by staying in the yellow region of Fig. 5.1. We call this *normal execution with bi-directional propagation*, since (only) unit propagation goes between solvers.

Speculation What sets SPECSMS apart is speculation: at any non-deterministically chosen decision level i , SPECSMS can pause deciding on the main solver and activate the secondary solver (i.e., transition to state $\langle D^i, P \rangle$). During speculation, only the secondary solver makes decisions. Since the main solver does not have a full satisfying assignment to its clauses, the secondary solver propagates assignments to the main solver and vice-versa.

Speculation terminates when the secondary solver S_s either: (1) hits a conflict that cannot be resolved by inter-modular conflict analysis; (2) hits a conflict below decision level i ; or (3) finds a satisfying assignment to Φ_s .

Case (1) is most interesting, and is what makes SPECSMS differ from SMS. Note that a conflict clause is not resolved by inter-modular conflict analysis only if it depends on an external literal on the trail of S_s that cannot be explained by an interface clause from S_m . This is possible when both S_m and S_s have partial assignments during speculation. So the conflict might depend on the *local* decisions of S_m . This cannot be communicated to S_s using only interface variables.

Refinement In SPECSMS, this is handled by modifying the REASON method in the solvers to fail (i.e., return ext) whenever **AnalyzeFinal** returns a non-interface clause. Additionally, the literal on which **AnalyzeFinal** failed is recorded in a global variable *refineLit*. This is shown in Alg. 14. The inter-modular conflict analysis is modified to exit early whenever REASON fails to produce a justification. At this

point, SPECSMS exits speculation, returns to the initial state $\langle P, D^0 \rangle$, both solvers back-jump to decision level i at which speculation was initiated, and S_m is forced to decide on *refineLit*.

We call this transition a *refinement* because the partial assignment of the main solver S_m (which we view as an *abstraction*) is updated (a.k.a., refined) based on the information that was not available to it (namely, a conflict with a set of decisions in the secondary solver S_s). Since *refineLit* was not decided on in S_m prior to speculation, deciding on it is a new decision that ensures progress in S_m . The next speculation is possible only under strictly more decisions in S_m than before, or when S_m back-jumps and flips an earlier decision.

We illustrate the refinement process on a simple example:

Example 13 ² Consider the query $\langle \Phi_s, \Phi_m \rangle$ with:

$\Phi_s(i, j, k, z):$	$\Phi_m(a, i, j, k):$
$\bar{z} \vee \bar{i} \quad (3)$	$\bar{a} \vee i \vee \bar{j} \quad (1)$
$i \vee j \vee \bar{k} \quad (4)$	$j \vee k \quad (2)$

First, S_m decides a (at level 1), which causes no propagations. Then, SPECSMS enters speculative mode, transitions to $\langle D^1, P \rangle$ and starts making decisions in S_s . S_s decides z and calls PROPAGATEALL. Afterwards, the trails for S_m and S_s are as follows:

S_m	$a @ 1$	$null @ 2$	$\bar{i} \text{ (ext)}$	$\bar{j} (1)$	$k (2)$
S_s	$null @ 1$	$z @ 2$	$\bar{i} (3)$	$\bar{j} \text{ (ext)}$	$k \text{ (ext)}$

where $x @ i$ denotes that literal x is decided at level i , and $x (r)$ denotes that literal x is propagated using a reason clause r , or due to the other solver (if $r = \text{ext}$). A conflict is hit in S_s in clause (4). Inter-modular conflict analysis begins. S_s first asks for the reason for k , which is clause (2) in S_m . This clause is copied to S_s . Note that unlike SMS, clauses can move from S_m to S_s . The new conflict to be analyzed is $(i \vee j \vee \bar{j})$. Now the reason for $\bar{j}$ is asked of S_m . In this case, S_m cannot produce a clause over shared variables to justify $\bar{j}$, so conflict analysis fails with *refineLit* = j . This causes SPECSMS to exit speculation mode and move to state $\langle P, D^0 \rangle$ and S_m must decide variable j before speculating again. In this case either decision on j results in $\langle \Phi_s, \Phi_m \rangle$ being sat. ■

In addition to refining when conflict analysis fails, SPECSMS also has the ability to refine non-deterministically. That is, at any point during speculation, S_s can decide

²In this chapter, we use $\bar{i}$ to mean the negation of boolean literal i

Algorithm 14: The REASON method in modular SAT solvers inside SPECSMS

```

1 function Reason(lit):
2   if reason[lit] = ext then
3      $c \leftarrow \text{other.AnalyzeFinal}(\text{lit})$ 
4     if  $\exists v \in c :: v \notin I$  then
5        $\text{refineLit} \leftarrow \text{lit}$ 
6     return ext
7   ADDCLAUSE(c)
8   reason[lit]  $\leftarrow c$ 
9   return reason[lit]

```

to stop speculation, back-jump to the decision level from which it started speculation, and choose any interface literal as *refineLit*.

Case (2) is similar to what happens in SMS when a conflict is detected in S_s . The reason for the conflict is below level i which is below the level of any decision of S_s . Since decision levels below i are treated as assumptions in S_s , calling **AnalyzeFinal** in S_s returns an interface clause c that blocks the current assignment in S_m . The clause c is added to S_m . The solvers back-jump to the smallest decision level j that makes c an asserting clause in S_m . Finally, SPECSMS moves to $\langle P, D^0 \rangle$.

Validation Case (3), like Case (1), is unique to SPECSMS. While all clauses of S_s are satisfied, the current assignment might not satisfy all clauses of S_m . Thus, SPECSMS enters *validation* by switching to the configuration $\langle F, D^M \rangle$, where M is the current decision level. Thus, S_m becomes active and starts deciding and propagating. This continues, until one of two things happen: (3a) S_m extends the assignment to satisfy all of its clauses, or (3b) a conflict that cannot be resolved with inter-modular conflict analysis is found. In the case (3a), SPECSMS transitions to $\langle F, F \rangle$ and declares that $\langle \Phi_m, \Phi_s \rangle$ is sat. The case (3b) is handled exactly the same as Case (1) – the literal on the trail without a reason is stored in *refineLit*, SPECSMS moves to $\langle P, D^0 \rangle$, backjumps to the level in which speculation was started, and S_m is forced to decide on *refineLit*.

Theorem 10 SPECSMS terminates. If it reaches the state $\langle F, F \rangle$, then $\Phi_s \wedge \Phi_m$ is satisfiable and the join of the trails of $\langle S_s, S_m \rangle$ is a satisfying assignment. If it reaches the state *unsat*, $\Phi_s \wedge \Phi_m$ is unsatisfiable. $\square$

5.3 Validation and interpolation

In this section, we augment SPECSMS with an interpolation procedure. To this end, we first introduce modular DRUP proofs, which are generated from SPECSMS in a natural way. We then present an algorithm for extracting an interpolant from a modular trimmed DRUP proof in the spirit of [58].

5.3.1 DRUP proofs for modular SAT

Modular DRUP proofs – a form of clausal proofs [50] – extend (monolithic) DRUP proofs [59]. A DRUP proof [59] is a sequences of steps, where each step either asserts a clause, deletes a clause, or adds a new Reverse Unit Propagation (RUP) clause. Given a set of clauses Γ , a clause cls is an RUP for Γ , written $\Gamma \vdash_{UP} cls$, if cls follows from Γ by unit propagation [49]. For a DRUP proof π , let $\text{ASSERTED}(\pi)$ denote all clauses of the asserted commands in π , then π shows that all RUP clauses of π follow from $\text{ASSERTED}(\pi)$. If π contains a $\perp$ clause, then π certifies $\text{ASSERTED}(\pi)$ is unsat.

A Modular DRUP proof is a sequence of clause addition and deletion steps, annotated with indices idx (**m** or **s**). Intuitively, steps with the same index must be validated together (within the same module idx), and steps with different indices may be checked independently. The steps are:

1. $(\text{asserted}, idx, cls)$ denotes that cls is asserted in idx ,
2. (rup, idx, cls) denotes adding RUP clause cls to idx ,
3. $(\text{cp}(src), dst, cls)$ denotes copying a clause cls from src to dst , and
4. (del, idx, cls) denotes removing clause cls from idx .

We denote the prefix of length k of a sequence of steps π by π^k . Given a sequence of steps π and a formula index idx , we use $\text{act_clauses}(\pi, idx)$ to denote the set of active clauses with index idx . Formally,

$$\begin{aligned} \{cls \mid \exists c_j \in \pi \cdot \\ (c_j = (t, idx, cls) \wedge (t = \text{asserted} \vee t = \text{rup} \vee t = \text{cp}(-))) \\ \wedge \neg \exists c_k \in \pi \cdot k > j \wedge c_k = (\text{del}, idx, cls)\} \end{aligned}$$

A sequence of steps $\pi = c_1, \dots, c_n$ is a *valid modular DRUP proof* iff for each $c_i \in \pi$:

1. if $c_i = (\text{rup}, idx, cls)$ then $\text{act_clauses}(\pi^i, idx) \vdash_{UP} cls$,

seq	step	to	clause
1	asserted	m	$\neg s_1 \Rightarrow lb_1$
2	asserted	m	$\neg s_1 \Rightarrow \neg lb_1$
3	asserted	s	$(s_1 \wedge la_1) \Rightarrow s_2$
4	asserted	s	$(s_1 \wedge \neg la_1) \Rightarrow s_2$
5	asserted	m	$(s_2 \wedge lb_2) \Rightarrow s_3$
6	asserted	m	$(s_2 \wedge \neg lb_2) \Rightarrow s_3$
7	asserted	s	$(s_3 \wedge la_2) \Rightarrow s_4$
8	asserted	s	$(s_3 \wedge \neg la_2) \Rightarrow s_4$
9	asserted	m	$s_4 \Rightarrow lb_3$
10	asserted	m	$s_4 \Rightarrow \neg lb_3$
11	rup	m	s_1
12	rup	m	$\neg s_4$
13	rup	m	$s_2 \Rightarrow s_3$
14	cp(m)	s	$s_2 \Rightarrow s_3$
15	rup	s	$s_3 \Rightarrow s_4$
16	rup	s	$s_1 \Rightarrow s_4$
17	cp(s)	m	$s_1 \Rightarrow s_4$
18	rup	m	$\perp$

Figure 5.2: An example of a modular DRUP proof. Clauses are written in human-readable form as implications, instead of in the DIMACS format.

2. if $c_i = (\text{cp}(\text{idx}), \neg, \text{cls})$ then $\text{act_clauses}(\pi^i, \text{idx}) \vdash_{UP} \text{cls}$, and
3. $c_{|\pi|}$ is either $(\text{rup}, \text{m}, \perp)$ or $(\text{cp}(\text{s}), \text{m}, \perp)$.

Let $\text{ASSERTED}(\pi, \text{idx})$ be the set of all asserted clauses in π with index idx .

Theorem 11 *If π is a valid modular DRUP proof, then $\text{ASSERTED}(\pi, \text{s}) \wedge \text{ASSERTED}(\pi, \text{m})$ is unsatisfiable.* $\square$

Modular DRUP proofs may be validated with either one or two solvers. To validate with one solver we convert the modular proof into a monolithic one (i.e., where the steps are asserted, rup, and del). Let MODDRUP2DRUP be a procedure that given a modular DRUP proof π , returns a DRUP proof π' that is obtained from π by (a) removing idx from all the steps; (b) removing all cp steps; (c) removing all del steps.

Note that del steps are removed for simplicity, otherwise it is necessary to account for deletion of copied and non-copied clauses separately.

Lemma 17 *If π is a valid modular DRUP proof then $\pi' = \text{MODDRUP2DRUP}(\pi)$ is a valid DRUP proof.* $\square$

Modular validation is done with two monolithic solvers working in lock step: (asserted, cls , idx) steps are added to the idx solver; (rup, idx , cls) steps are validated locally in solver idx using all active clauses (asserted, copied, and rup); and for (cp(src), dst , cls) steps, cls is added to dst but not validated in it, and cls is checked to exist in the src solver.

From now on, we consider only valid proofs. We say that a (valid) modular DRUP proof π is a proof of unsatisfiability of $\Phi_s \wedge \Phi_m$ if $\text{ASSERTED}(\pi, s) \subseteq \Phi_s$ and $\text{ASSERTED}(\pi, m) \subseteq \Phi_m$ (inclusion here refers to the sets of clauses).

SPECSMS produces modular DRUP proofs by logging the clauses that are learnt, deleted, and copied between solvers. Note that in SMS clauses may only be copied from S_s to S_m , but in SPECSMS they might be copied in both directions.

Theorem 12 *Let Φ_s and Φ_m be two Boolean formulas s.t. $\Phi_s \wedge \Phi_m \models \perp$. SPECSMS produces a valid modular DRUP proof for unsatisfiability of $\Phi_s \wedge \Phi_m$.* $\square$

Algorithm 15: Trimming a modular DRUP proof.

Input: Solver instances S_s, S_m with the empty clause on the trail, and a modular clausal proof $\pi = c_1, \dots, c_n$.

Output: A proof π' s.t. all steps are core.

```

1  $\pi' = \emptyset$ 
2  $M_s, M_m \leftarrow \{\perp\}, \emptyset$  // Relevant clauses
3 for  $i = n$  to 0 do
4   match  $c_i$  with  $(type, idx, cls)$ 
5   if  $cls \notin M_{idx}$  then
6     continue
7   if  $type = del$  then
8      $S_{idx}.Revive(cls)$ 
9     continue
10   $\pi'.append(c_i)$ 
11  if  $type = rup$  then
12     $S_{idx}.CHK\_RUP(cls, M_{idx})$ 
13  else if  $type = cp(src)$  then
14     $S_{idx}.Delete(cls)$ 
15     $M_{src}.add(cls)$ 
16   $\pi'.reverse()$ 
17 function  $solver::CHK\_RUP(cls, M)$ :
18   if  $IsOnTrail(cls)$  then
19      $UndoTrail(cls)$ 
20      $Delete(cls)$ 
21      $SaveTrail()$ 
22      $Enqueue(\neg cls)$ 
23      $r \leftarrow Propagate()$ 
24      $ConflictAnalysis(r, M)$ 
    /* Updates  $M$  with conflict clauses */
25    $RestoreTrail()$ 
```

Algorithm 16: Interpolating a modular DRUP proof.

Input: Propositional formulas $\langle \Phi_0, \Phi_1 \rangle$

Input: A modular trimmed DRUP proof $\pi = c_1, \dots, c_n$ of unsatisfiability of $\Phi_0 \wedge \Phi_1$

Output: An interpolant itp s.t. $\Phi_0 \Rightarrow itp$ and $itp \wedge \Phi_1 \models \perp$

```

1  $S_s, S_m := SAT\_SOLVER()$ 
2  $itp := \top$ 
3 for  $i = 0$  to  $n$  do
4   begin
5     switch  $c_i$  do
6     case  $(asserted, s, cls)$  do
7        $sup(cls) \leftarrow \top$ 
8     case  $(cp(m), s, cls)$  do
9        $sup(cls) \leftarrow cls$ 
10    case  $(rup, s, cls)$  do
11       $M := \emptyset$ 
12       $S_s.CHK\_RUP(cls, M)$ 
13       $sup(cls) \leftarrow \{sup(c) \mid c \in M\}$ 
14    case  $(cp(s), m, cls)$  do
15       $itp \leftarrow itp \wedge (sup(cls) \Rightarrow cls)$ 
16     $S_{c_i.idx}.add(cls)$ 
```

Trimming modular DRUP proofs. A step in a modular DRUP proof π is *core* if removing it invalidates π . Under this definition, *del* steps are never core since removing them does not affect validation. Alg. 15 shows an algorithm to trim modular DRUP proofs based on backward validation. The input are two modular solvers S_m and S_s in a final conflicting state, and a valid modular DRUP proof $\pi = c_1, \dots, c_n$. The output is a trimmed proof π' s.t. all steps of π' are core.

We assume that the reader is familiar with MiniSAT [34] and use the following solver methods: **Propagate**, exhaustively applies unit propagation (UP) rule by resolving all unit clauses; **ConflictAnalysis** analyzes the most recent conflict and marks which clauses are involved in the conflict; **IsOnTrail** checks whether a clause is an antecedent

of a literal on the trail; **Enqueue** enqueues one or more literals on the trail; **IsDeleted**, **Delete**, **Revive** check whether a clause is deleted, delete a clause, and add a previously deleted clause, respectively; **SaveTrail**, **RestoreTrail** save and restore the state of the trail.

Alg. 15 processes the steps of the proof backwards, rolling back the states of the solvers. M_{idx} marks which clauses were relevant to derive clauses in the current suffix of the proof. While the proof is constructed through inter-modular reasoning, the trimming algorithm processes each of the steps in the proof completely locally. During the backward construction of the trimmed proof, steps that include unmarked clauses are ignored (and, in particular, not added to the proof). For each (relevant) rup step, function **CHK_RUP**, using **ConflictAnalysis**, adds clauses to M . **del** steps are never added to the trimmed proof, but the clause is revived from the solver. For **cp** steps, if the clause was marked, it is marked as used for the solver it was copied from and the step is added to the proof. Finally, asserted clauses that were marked are added to the trimmed proof. Note that, as in [58], proofs may be trimmed in different ways, depending on the strategy for **ConflictAnalysis**.

The following theorem states that trimming preserves validity of the proof:

Theorem 13 *Let Φ_s and Φ_m be two formulas such that $\Phi_s \wedge \Phi_m \models \perp$. If π is a modular DRUP proof produced by solvers S_s and Φ_m for $\Phi_s \wedge \Phi_m$, then a trimmed proof π' by Alg. 15 is also a valid modular DRUP proof for $\Phi_s \wedge \Phi_m$. $\square$*

Fig. 5.2 shows a trimmed proof after SPECSMS is executed on $\langle \psi_0, \psi_1 \rangle$ such that $\psi_0 \triangleq ((s_1 \wedge la_1) \Rightarrow s_2) \wedge ((s_1 \wedge \neg la_1) \Rightarrow s_2) \wedge ((s_3 \wedge la_2) \Rightarrow s_4) \wedge ((s_3 \wedge \neg la_2) \Rightarrow s_4)$ and $\psi_1 \triangleq (\neg s_1 \Rightarrow lb_1) \wedge (\neg s_1 \Rightarrow \neg lb_1) \wedge ((s_2 \wedge lb_2) \Rightarrow s_3) \wedge ((s_2 \wedge \neg lb_2) \Rightarrow s_3) \wedge (s_4 \Rightarrow lb_3) \wedge (s_4 \Rightarrow \neg lb_3)$.

5.3.2 Interpolation

Given a modular DRUP proof π of unsatisfiability of $\Phi_s \wedge \Phi_m$, we give an algorithm to compute an interpolant of $\Phi_s \wedge \Phi_m$. For simplicity of the presentation, we assume that π has no deletion steps; this is the case in trimmed proofs, but we can also adapt the interpolation algorithm to handle deletions by keeping track of active clauses.

Our interpolation algorithm relies only on the clauses copied between the modules. Notice that whenever a clause is copied from module i to module j , it is implied by all the clauses in Φ_i together with all the clauses that have been copied from module j . We refer to clauses copied from S_m to S_s as *backward* clauses and clauses copied from S_s to S_m as *forward* clauses. The conjunction of forward clauses is unsatisfiable with S_m . This is because, in the last step of π , $\perp$ is added to S_m , either through rup or

by $\text{cp} \perp$ from S_s . Since all the clauses in module $\mathbf{m}$ are implied by $\Phi_{\mathbf{m}}$ together with forward clauses, this means that the conjunction of forward clauses is unsatisfiable with $\Phi_{\mathbf{m}}$. In addition, all forward clauses were learned in module $\mathbf{s}$, with support from backward clauses. This means that every forward clause is implied by Φ_s together with the subset of the backward clauses used to derive it. Intuitively, we should therefore be able to learn an interpolant with the structure: backward clauses imply forward clauses.

Alg. 16 describes our interpolation algorithm. It traverses a modular DRUP proof forward. For each clause cls learned in module $\mathbf{s}$, the algorithm collects the set of backward clauses used to learn cls . This is stored in the sup datastructure — a mapping from clauses to sets of clauses. Finally, when a forward clause c is copied, it adds $\text{sup}(c) \Rightarrow c$ to the interpolant.

Example 14 We illustrate our algorithm using the modular DRUP proof from Fig. 5.2. On the first cp step ($\text{cp}(\mathbf{m}), \mathbf{s}, s_2 \Rightarrow s_3$), the algorithm assigns the sup for clause $s_2 \Rightarrow s_3$ as itself (line 9). The first clause learnt in module $\mathbf{s}$, ($\text{rup}, \mathbf{s}, s_3 \Rightarrow s_4$), is derived from just the clauses in module $\mathbf{s}$ and no backward clauses. Therefore, after RUP, our algorithm sets $\text{sup}(s_3 \Rightarrow s_4)$ to $\top$ (line 13). The second clause learnt in module $\mathbf{s}$, $s_1 \Rightarrow s_4$, is derived from module $\mathbf{s}$ with the support of the backward clause $s_2 \Rightarrow s_3$. Therefore, $\text{sup}(s_1 \Rightarrow s_4) = \{s_2 \Rightarrow s_3\}$. When this clause is copied forward to module 1, the algorithm updates the interpolant to be $(s_2 \Rightarrow s_3) \Rightarrow (s_1 \Rightarrow s_4)$. ■

Next, we formalize the correctness of the algorithm. Let $L_B(\pi) = \{cls \mid (\text{cp}(\mathbf{m}), \mathbf{s}, cls) \in \pi\}$ be the set of clauses copied from module $\mathbf{m}$ to $\mathbf{s}$ and $L_F(\pi) = \{cls \mid (\text{cp}(\mathbf{s}), \mathbf{m}, cls) \in \pi\}$ be clauses copied from module $\mathbf{s}$ to $\mathbf{m}$. From the validity of modular DRUP proofs, we have that:

Lemma 18 *For any step $c_i = (\text{cp}(\mathbf{s}), \mathbf{m}, cls) \in \pi$, $(L_B(\pi^i) \wedge \Phi_s) \Rightarrow cls$ and for any step $c_j = (\text{cp}(\mathbf{m}), \mathbf{s}, cls) \in \pi$, $(L_F(\pi^j) \wedge \Phi_m) \Rightarrow cls$.* □

For any clause cls copied from one module to the other, we use the shorthand $\sharp(cls)$ to refer to the position of the copy command in the proof π . That is, $\sharp(cls)$ is the smallest k such that $c_k = (\text{cp}(i), j, cls) \in \pi$. The following is an invariant in a valid modular DRUP proof:

Lemma 19

$$\forall cls \in L_F(\pi) \cdot (\Phi_{\mathbf{m}} \wedge (L_F(\pi^{\sharp(cls)}))) \Rightarrow L_B(\pi^{\sharp(cls)}) \quad \square$$

These properties ensure that adding $L_B(\pi^{\sharp(cls)}) \Rightarrow cls$ for every forward clause cls results in an interpolant. Alg. 16 adds $(\text{sup}(cls) \Rightarrow cls)$ as an optimization. Correctness is preserved since $\text{sup}(cls)$ is a subset of $L_B(\pi^{\sharp(cls)})$ that together with Φ_s suffices to derive cls (formally, $\text{sup}(cls) \wedge \Phi_s \vdash_{UP} cls$).

Theorem 14 *Given a modular DRUP proof π for $\Phi_s \wedge \Phi_m$, $itp \triangleq \{\sup(c) \Rightarrow c \mid c \in L_F(\pi)\}$ is an interpolant for $\langle \Phi_s, \Phi_m \rangle$. $\square$*

PROOF Since all copy steps are over interface variables, the interpolant is also over interface variables. By Lem. 18 (and the soundness of sup optimization), $\Phi_s \Rightarrow itp$. Next, we prove that $(\Phi_m \wedge itp) \Rightarrow \perp$. From Lem. 19, we have that for all $c \in L_F(\pi)$, $(\Phi_m \wedge L_F(\pi^{\sharp(c)})) \Rightarrow \sup(c)$. Therefore, $(\Phi_m \wedge L_F(\pi^{\sharp(c)}) \wedge (\sup(c) \Rightarrow c)) \Rightarrow c$ $\blacksquare$

It is much simpler to extract interpolants from modular DRUP proofs than from arbitrary DRUP proofs. This is not surprising since the interpolants capture exactly the information that is exchanged between solvers. The interpolants are not in CNF, but can be converted to CNF after extraction.

5.4 Heuristics for guiding specSMS

Theoretically, speculation makes SPECSMS more powerful than SMS and IC3/PDR. However, in practice, deciding when to enter speculation has a major impact on the performance of SPECSMS. If the speculation is too greedy, SPECSMS performs poorly on examples where the main module is easy to solve. Similarly, if the speculation is too lazy, SPECSMS performs poorly on problems in which any solution to the secondary module makes the main module easy to solve. We illustrate this trade-off using an example.

Example 15 Consider a modular query: $\langle \gamma_{in}(\ell, x, in), \gamma_{SHA-1}(in, x, out) \rangle$, where x is an 512-bit vector, ℓ is a 160-bit vector, $chks_i$ are 512-bit vector, and the remaining variables are the same as in ψ_{in} and ψ_{SHA-1} , and

$$\begin{aligned} \gamma_{in} &\triangleq \text{SHA-1}_{circ}(x, \ell) \wedge \\ &\quad ((\ell = chks_0 \wedge in = msg_0) \vee (\ell = chks_1 \wedge in = msg_1) \vee \\ &\quad (\ell = chks_2 \wedge in = msg_2) \vee (\ell = chks_3 \wedge in = msg_3)) \\ \gamma_{SHA-1} &\triangleq (x = 1 \vee x = 4) \wedge \text{SHA-1}_{circ}(in, out) \wedge out = shaVal \end{aligned}$$

This is an example where bi-directional search is necessary to efficiently solve the query. If deciding only on γ_{SHA-1} , we encounter the hard problem of inverting SHA-1_{circ} , if deciding in γ_{in} , we encounter the same problem, since an assignment for x needs to be found, based on the four values for ℓ . Therefore, neither immediate nor late speculation makes SPECSMS efficient on the problem. The ideal strategy here is to speculate after an assignment to x , to simplify γ_{in} . $\blacksquare$

time (s) – sat			time (s) – unsat		
# rounds	SMS	SPECSMS	# rounds	SMS	SPECSMS
16	0.86	0.94	16	1.09	0.93
21	–	0.49	21	–	1.17
26	–	2.93	26	–	1.95
31	–	1.33	31	–	2.06
36	–	1.35	36	–	2.13
40	–	1.56	40	–	2.64

Table 5.1: Solving time with a timeout of 600s.

Ideally, we would like to speculate when the current modular query is too hard for the solver. As a proxy for hardness, we measure the number of conflicts the SAT solver hits. We first speculate when the main solver hits a predetermined number of conflicts. We then exponentially widen the number of conflicts between speculations. Exiting from speculation is just as important as entering speculation: the secondary solver might also get stuck in solving its module. Therefore, we use the same heuristic in the secondary solver to exit speculation.

While this is a simple heuristic, we found it to be useful in our benchmarks. The best strategy for speculation is problem-dependent. We leave development of a robust heuristic for future work.

5.5 Implementation and Validation

We implemented SPECSMS (and SMS) inside the extensible SAT-solver of Z3 [85]³. For SMS, we simply disable speculation.

We have validated SPECSMS on a set of handcrafted benchmarks, based on Ex. 11. Each benchmark is of the form $\langle \psi_{in}(\ell, in), \psi_{SHA-1}(in, out) \rangle$, where ℓ is a 2-bit vector, in is a 512-bit vector (shared), out is 160-bit vector. ψ_{in} encodes that there are four possible messages:

$$\begin{aligned} \psi_{in} \triangleq & (\ell = 0 \wedge in = msg_0) \vee (\ell = 1 \wedge in = msg_1) \vee \\ & (\ell = 2 \wedge in = msg_2) \vee (\ell = 3 \wedge in = msg_3) \end{aligned}$$

and $\psi_{SHA-1}(in, out)$ encodes the SHA-1 circuit together with some hash:

$$\psi_{SHA-1} \triangleq (SHA-1_{circ}(in) \wedge out = shaVal)$$

³<https://github.com/hgvk94/z3/tree/psms>.

In the first set of experiments, we check sat queries by generating one msg_i in ψ_{in} that produces $shaVal$. In the second set, we check unsat queries, by ensuring that no msg_i produces $shaVal$. To evaluate performance, we make ψ_{SHA-1} harder to solve by increasing the number of rounds of SHA-1 circuit encoded in the $SHA-1_{circ}$ clauses. We used SAT-encoding [87]⁴ to generate the $SHA-1_{circ}$ with the different number of rounds (SAT-encoding supports 16 to 40 rounds).

We use the heuristic described in Sec. 5.4 to decide when to enter and exit speculation. Thus, SPECSMS switches modules when it hits too many conflicts in the module. In contrast, SMS only switches to the secondary solver after finding a full satisfying assignment in the main solver.

Results for each set of the queries are shown in Tab. 5.1. Column “# rounds” shows the number of SHA-1 rounds encoded in ψ_{SHA-1} . The problems quickly become too hard for SMS. At the same time, SPECSMS solves all the queries quickly. Furthermore, the run-time of SPECSMS appears to grow linearly with the number of rounds.

The experiments validate our claim that switching between modules is quite effective in solving the problem. As expected, SMS gets stuck in inverting the SHA-1 function. It cannot make progress without using information from the secondary module. In contrast, SPECSMS switches to the secondary module once it finds that solving $SHA-1_{circ}(in)$ is hard. Note that, in this problem, the ideal strategy is to speculate eagerly and then branch on all the ℓ variables. However, SPECSMS spend some time solving $SHA-1_{circ}(in)$. It only switches to the secondary module when it hits many conflicts in $SHA-1_{circ}(in)$.

5.6 Conclusion and Future Work

Modular SAT-solving is crucial for efficient SAT-based unbounded Model Checking. Existing techniques, embedded in IC3/PDR and extended in SMS [9], trade the efficiency of the solver for the simplicity of conflict resolution. We propose a new modular SAT-solver, called SPECSMS, that extends SMS with truly bi-directional reasoning. We show that it is provably better than SMS (and, therefore, IC3/PDR). We implement SPECSMS in Z3, extend it with DRUP-style [59] proofs, and proof-based interpolation. This work is an avenue to future efficient SAT- and SMT-based Model Checking algorithms.

We rely on a simple heuristic to guide SPECSMS when to start speculation and exit speculation. This is sufficient to show the power of bi-directional reasoning over uni-directional reasoning on our benchmarks. However, other application domains might need more complicated heuristics to make this decision. In the future, we plan

⁴Available at <https://github.com/saeednj/SAT-encoding>.

to explore guiding speculation using similar strategy used for guiding restarts in a modern CDCL SAT-solver [11].

A much earlier version of speculation, called *weak abstraction*, is implemented in SPACER. Since SPACER extends IC3/PDR to SMT, the choice of speculation is based on theory reasoning. Speculation starts when the main solver is satisfied modulo some theories (e.g., Linear Real Arithmetic or Weak Theory of Arrays). Speculation often prevents SPACER from being stuck in any one SMT query. However, SPACER has no inter-modular propagation and no *refinement*. If *validation* fails, speculation is simply disabled and the query is tried again without it. We hope that extending SPECSMS to theories will make SPACER heuristics much more flexible and effective.

DPLL(T)-style [44] SMT-solvers can be seen as modular SAT-solvers where the main module is a SAT solver and the secondary solver is a theory solver (often EUF-solver that is connected to other theory solvers such as a LIA solver). This observation is credited as an intuition for SMS [9]. In modern SMT-solvers, all decisions are made by the SAT-solver. For example, if a LIA solver wants to split on a bound of a variable x , it first adds a clause $(x \leq (b - 1) \vee x \geq b)$, where b is the desired bound, to the SAT-solver and then lets the SAT-solver branch on the clause. SPECSMS extends this interaction by allowing the secondary solver (i.e., the theory solver) to branch without going back to the main solver. Control is returned to the main solver only if such decisions tangle local decisions of the two solvers. We hope that the core ideas of SPECSMS can be lifted to SMT and allow more flexibility in the interaction between the DPLL-core and theory solvers. Similar interactions between a core SAT solver and a secondary external solver can also be found in programmatic SAT solvers [42]. Here, the user encodes domain knowledge as inference rules (a program) to guide a SAT solver during its search. These programs detect whether the partial assignment of the SAT solver is in conflict and generates a conflict clause to be added to the SAT solver. SPECSMS architecture allows programmatic SAT solvers to use richer programs that maintain state and make decisions. It is interesting to see how beneficial such programs are to programmatic SAT solvers.

Chapter 6

Conclusions

This thesis presented a series of algorithmic improvements to the state of the art in CHC solving. We first looked at the problem of generalizing lemmas to construct a solution. To overcome the limitations of local generalization inside CHC solvers, we proposed global guidance and showed that it works well on a wide variety of benchmarks and under different local generalization. To broaden the applicability of CHC solvers, we looked at CHC solving modulo the theory of algebraic datatypes and recursive functions. We used a combination of abstraction-refinement steps and CHC encoding of recursive functions, to overcome undecidability of the underlying theory. To improve the efficiency of CHC solving, we study two key subproblems that arise during the solving process: approximations of quantifier elimination and modular solving. We implement better solutions to these problems. Ideas from this thesis have been implemented on top of the SPACER CHC solver inside the Z3 SMT solver. Some of them have been integrated to the main branch of Z3 and are being used in various model checking applications.

This thesis opens doors to many future research directions. We discuss two possibilities here. In the era of Machine Learning and generative AI (ML), an obvious question is how to use ML algorithms for global guidance. ML algorithms have proven to be very good at identifying patterns in large datasets. Recent work [74] has used ML to identify patterns in lemmas and guide inductive generalization inside SPACER. It is very interesting to see if ML models can learn new global guidance rules. A particularly interesting direction is coming up with global guidance rules to do symmetry breaking in distributed system verification.

Another interesting direction is building an *extendable* CHC solver. Two key ingredients of our extension to the theory of ADTs and RDFs were: (1) abstracting complicated theory operations using EUF and (2) having an abstraction-refinement loop inside the modular model checking core of the algorithm. Lifting these ideas to

be applicable to other theories as well, gives us a new kind of *extendable* CHC solver. We envision a CHC solver that abstracts all theory operators using UF and works over the core theory of EUF. Such a solver only applies theory axioms on demand during refinement. The solver is applicable to all theories and makes progress even when the background theory is undecidable (because of the UF abstraction). The Speculative SMS algorithm in Chp. 5 takes a first step towards such a solver. Lifting the algorithm from a boolean modular SAT solving algorithm to an infinite state model checking algorithm is an obvious necessary next step. Another challenge to be solved is building a CHC solver for the theory of EUF. The main hurdle there is the absence of qelim for EUF. Fortunately, the most precise over-approximation of qelim, called cover, is a good enough replacement of qelim for model checking algorithms [52]. Our QEL algorithm (Chp. 4) is a starting point to implement cover algorithms and its model based under-approximations. Furthermore, we can build off the egraph based combination of MBP algorithms to combine under-approximations of cover for various theories.

Bibliography

- [1] A. Albarghouthi and K. L. McMillan, “Beautiful interpolants”, in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, 2013, pp. 313–329. DOI: [10.1007/978-3-642-39799-8_22](https://doi.org/10.1007/978-3-642-39799-8_22). [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_22.
- [2] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina, “Lazy abstraction with interpolants for arrays”, in *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, N. Bjørner and A. Voronkov, Eds., ser. Lecture Notes in Computer Science, vol. 7180, Springer, 2012, pp. 46–61. DOI: [10.1007/978-3-642-28717-6_7](https://doi.org/10.1007/978-3-642-28717-6_7).
- [3] L. Alt, M. Blicha, A. E. J. Hyvärinen, and N. Sharygina, “SolCMC: Solidity Compiler’s Model Checker”, in *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, S. Shoham and Y. Vizel, Eds., ser. Lecture Notes in Computer Science, vol. 13371, Springer, 2022, pp. 325–338. DOI: [10.1007/978-3-031-13185-1_16](https://doi.org/10.1007/978-3-031-13185-1_16).
- [4] L. Alt and C. Reitwießner, “Smt-based verification of solidity smart contracts”, in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, T. Margaria and B. Steffen, Eds., ser. Lecture Notes in Computer Science, vol. 11247, Springer, 2018, pp. 376–388. DOI: [10.1007/978-3-030-03427-6_28](https://doi.org/10.1007/978-3-030-03427-6_28). [Online]. Available: https://doi.org/10.1007/978-3-030-03427-6_28.
- [5] K. Athanasiou, B. Cook, M. Emmi, C. MacCárthaigh, D. Schwartz-Narbonne, and S. Tasiran, “Sidetrail: Verifying time-balancing of cryptosystems”, in *Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers*, R. Piskac and P. Rümmer, Eds., ser. Lecture Notes in Computer Science, vol. 11294,

- Springer, 2018, pp. 215–228. DOI: [10.1007/978-3-030-03592-1_12](https://doi.org/10.1007/978-3-030-03592-1_12). [Online]. Available: https://doi.org/10.1007/978-3-030-03592-1_12.
- [6] J. Backes, P. Bolignano, B. Cook, *et al.*, “Semantic-based automated reasoning for AWS access policies using SMT”, in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, N. Bjørner and A. Gurfinkel, Eds., IEEE, 2018, pp. 1–9. DOI: [10.23919/FMCAD.2018.8602994](https://doi.org/10.23919/FMCAD.2018.8602994). [Online]. Available: <https://doi.org/10.23919/FMCAD.2018.8602994>.
 - [7] C. Barrett, P. Fontaine, and C. Tinelli, *The Satisfiability Modulo Theories Library (SMT-LIB)*, www.SMT-LIB.org, 2016.
 - [8] C. Barrett, A. Stump, and C. Tinelli, “The SMT-LIB Standard: Version 2.0”, in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening, Eds., 2010.
 - [9] S. Bayless, C. G. Val, T. Ball, H. H. Hoos, and A. J. Hu, “Efficient modular SAT solving for IC3”, in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, IEEE, 2013, pp. 149–156. [Online]. Available: <https://ieeexplore.ieee.org/document/6679404/>.
 - [10] F. Benoy, A. King, and F. Mesnard, “Computing convex hulls with a linear solver”, *TPLP*, vol. 5, no. 1-2, pp. 259–271, 2005. DOI: [10.1017/S1471068404002261](https://doi.org/10.1017/S1471068404002261). [Online]. Available: <https://doi.org/10.1017/S1471068404002261>.
 - [11] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, “CaDiCaL, Kissat, Para-cooba, Plingeling and Treengeling entering the SAT Competition 2020”, in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., ser. Department of Computer Science Report Series B, vol. B-2020-1, University of Helsinki, 2020, pp. 51–53.
 - [12] J. Birgmeier, A. R. Bradley, and G. Weissenbacher, “Counterexample to induction-guided abstraction-refinement (CTIGAR)”, in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, 2014, pp. 831–848. DOI: [10.1007/978-3-319-08867-9_55](https://doi.org/10.1007/978-3-319-08867-9_55). [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_55.
 - [13] N. Bjørner and A. Gurfinkel, “Property directed polyhedral abstraction”, in *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, 2015, pp. 263–281. DOI: [10.1007/978-3-662-46081-8_15](https://doi.org/10.1007/978-3-662-46081-8_15). [Online]. Available: https://doi.org/10.1007/978-3-662-46081-8_15.

- [14] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko, “Horn clause solvers for program verification”, in *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte, Eds., ser. Lecture Notes in Computer Science, vol. 9300, Springer, 2015, pp. 24–51. DOI: [10.1007/978-3-319-23534-9_2](https://doi.org/10.1007/978-3-319-23534-9_2).
- [15] N. Bjørner and M. Janota, “Playing with quantified satisfaction”, in *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015*, 2015, pp. 15–27. [Online]. Available: <http://www.easychair.org/publications/paper/252316>.
- [16] M. Blicha, A. E. J. Hyvärinen, J. Kofron, and N. Sharygina, “Decomposing Farkas interpolants”, in *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, 2019, pp. 3–20. DOI: [10.1007/978-3-030-17462-0_1](https://doi.org/10.1007/978-3-030-17462-0_1). [Online]. Available: https://doi.org/10.1007/978-3-030-17462-0%5C_1.
- [17] A. R. Bradley, “SAT-Based Model Checking without Unrolling”, in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, 2011, pp. 70–87. DOI: [10.1007/978-3-642-18275-4_7](https://doi.org/10.1007/978-3-642-18275-4_7). [Online]. Available: https://doi.org/10.1007/978-3-642-18275-4%5C_7.
- [18] P. E. Bulychov, E. V. Kostylev, and V. A. Zakharov, “Anti-unification algorithms and their applications in program analysis”, in *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, 2009, pp. 413–423. DOI: [10.1007/978-3-642-11486-1_35](https://doi.org/10.1007/978-3-642-11486-1_35). [Online]. Available: https://doi.org/10.1007/978-3-642-11486-1%5C_35.
- [19] A. Champion, T. Chiba, N. Kobayashi, and R. Sato, “ICE-Based Refinement Type Discovery for Higher-Order Functional Programs”, in *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, 2018, pp. 365–384. DOI: [10.1007/978-3-319-89960-2_20](https://doi.org/10.1007/978-3-319-89960-2_20). [Online]. Available: https://doi.org/10.1007/978-3-319-89960-2%5C_20.

- [20] B. E. Chang and K. R. M. Leino, “Abstract interpretation with alien expressions and heap structures”, in *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, R. Cousot, Ed., ser. Lecture Notes in Computer Science, vol. 3385, Springer, 2005, pp. 147–163. DOI: [10.1007/978-3-540-30579-8_11](https://doi.org/10.1007/978-3-540-30579-8_11).
- [21] N. Chong, B. Cook, K. Kallas, *et al.*, “Code-level model checking in the software development workflow”, in *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothmel and D. Bae, Eds., ACM, 2020, pp. 11–20. DOI: [10.1145/3377813.3381347](https://doi.org/10.1145/3377813.3381347). [Online]. Available: <https://doi.org/10.1145/3377813.3381347>.
- [22] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “IC3 modulo theories via implicit predicate abstraction”, in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, E. Ábrahám and K. Havelund, Eds., ser. Lecture Notes in Computer Science, vol. 8413, Springer, 2014, pp. 46–61. DOI: [10.1007/978-3-642-54862-8_4](https://doi.org/10.1007/978-3-642-54862-8_4). [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_4.
- [23] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “Infinite-state invariant checking with IC3 and predicate abstraction”, *Formal Methods in System Design*, vol. 49, no. 3, pp. 190–218, 2016. DOI: [10.1007/s10703-016-0257-4](https://doi.org/10.1007/s10703-016-0257-4). [Online]. Available: <https://doi.org/10.1007/s10703-016-0257-4>.
- [24] SV-COMP, *SV-COMP*, <https://sv-comp.sosy-lab.org/>.
- [25] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints”, in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, 1977*, pp. 238–252. DOI: [10.1145/512950.512973](https://doi.org/10.1145/512950.512973). [Online]. Available: <https://doi.org/10.1145/512950.512973>.
- [26] W. Craig, “Three uses of the herbrand-gentzen theorem in relating model theory and proof theory”, *J. Symb. Log.*, vol. 22, no. 3, pp. 269–285, 1957. DOI: [10.2307/2963594](https://doi.org/10.2307/2963594). [Online]. Available: <https://doi.org/10.2307/2963594>.
- [27] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, “Relational verification through horn clause transformation”, in *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, X. Rival, Ed., ser. Lecture Notes in Computer Science, vol. 9837,

- Springer, 2016, pp. 147–169. DOI: [10.1007/978-3-662-53413-7_8](https://doi.org/10.1007/978-3-662-53413-7_8). [Online]. Available: https://doi.org/10.1007/978-3-662-53413-7%5C_8.
- [28] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, “Removing algebraic data types from constrained horn clauses using difference predicates”, in *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, N. Peltier and V. Sofronie-Stokkermans, Eds., ser. Lecture Notes in Computer Science, vol. 12166, Springer, 2020, pp. 83–102. DOI: [10.1007/978-3-030-51074-9_6](https://doi.org/10.1007/978-3-030-51074-9_6). [Online]. Available: https://doi.org/10.1007/978-3-030-51074-9%5C_6.
 - [29] D. Detlefs, G. Nelson, and J. B. Saxe, “Simplify: A theorem prover for program checking”, *J. ACM*, vol. 52, no. 3, pp. 365–473, May 2005, ISSN: 0004-5411. DOI: [10.1145/1066100.1066102](https://doi.org/10.1145/1066100.1066102).
 - [30] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O’Hearn, “Scaling static analyses at facebook”, *Commun. ACM*, vol. 62, no. 8, pp. 62–70, 2019. DOI: [10.1145/3338112](https://doi.org/10.1145/3338112). [Online]. Available: <https://doi.org/10.1145/3338112>.
 - [31] B. Dutertre, “Yices 2.2”, in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, A. Biere and R. Bloem, Eds., ser. Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 737–744. DOI: [10.1007/978-3-319-08867-9_49](https://doi.org/10.1007/978-3-319-08867-9_49).
 - [32] B. Dutertre, “Solving Exists/Forall Problems with Yices”, in *Workshop on Satisfiability Modulo Theories*, 2015. [Online]. Available: <https://yices.csli.sri.com/papers/smt2015.pdf>.
 - [33] N. Eén, A. Mishchenko, and R. K. Brayton, “Efficient implementation of property directed reachability”, in *International Conference on Formal Methods in Computer-Aided Design, FMCAD ’11, Austin, TX, USA, October 30 - November 02, 2011*, P. Bjesse and A. Slobodová, Eds., FMCAD Inc., 2011, pp. 125–134. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2157675>.
 - [34] N. Eén and N. Sörensson, “An extensible sat-solver”, in *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, E. Giunchiglia and A. Tacchella, Eds., ser. Lecture Notes in Computer Science, vol. 2919, Springer, 2003, pp. 502–518. DOI: [10.1007/978-3-540-24605-3_37](https://doi.org/10.1007/978-3-540-24605-3_37). [Online]. Available: https://doi.org/10.1007/978-3-540-24605-3%5C_37.
 - [35] H. B. Enderton, *A Mathematical Introduction to Logic*. New York, Academic Press, 1972.

- [36] Z. Esen and P. Rümmer, “Tricera: Verifying C programs using the theory of heaps”, in *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, A. Griggio and N. Rungta, Eds., IEEE, 2022, pp. 380–391. DOI: [10.34727/2022/ISBN.978-3-85448-053-2_45](https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_45). [Online]. Available: https://doi.org/10.34727/2022/isbn.978-3-85448-053-2%5C_45.
- [37] G. Fedyukovich, S. J. Kaufman, and R. Bodík, “Sampling invariants from frequency distributions”, in *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, D. Stewart and G. Weissenbacher, Eds., IEEE, 2017, pp. 100–107. DOI: [10.23919/FMCAD.2017.8102247](https://doi.org/10.23919/FMCAD.2017.8102247). [Online]. Available: <https://doi.org/10.23919/FMCAD.2017.8102247>.
- [38] G. Fedyukovich, S. J. Kaufman, and R. Bodík, “Learning inductive invariants by sampling from frequency distributions”, *Formal Methods Syst. Des.*, vol. 56, no. 1, pp. 154–177, 2020. DOI: [10.1007/s10703-020-00349-x](https://doi.org/10.1007/s10703-020-00349-x). [Online]. Available: <https://doi.org/10.1007/s10703-020-00349-x>.
- [39] J. Filliâtre and A. Paskevich, “Why3 - where programs meet provers”, in *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, M. Felleisen and P. Gardner, Eds., ser. Lecture Notes in Computer Science, vol. 7792, Springer, 2013, pp. 125–128. DOI: [10.1007/978-3-642-37036-6_8](https://doi.org/10.1007/978-3-642-37036-6_8). [Online]. Available: https://doi.org/10.1007/978-3-642-37036-6%5C_8.
- [40] C. Flanagan, R. Joshi, and K. R. M. Leino, “Annotation inference for modular checkers”, *Inf. Process. Lett.*, vol. 77, no. 2-4, pp. 97–108, 2001. DOI: [10.1016/S0020-0190\(00\)00196-4](https://doi.org/10.1016/S0020-0190(00)00196-4). [Online]. Available: [https://doi.org/10.1016/S0020-0190\(00\)00196-4](https://doi.org/10.1016/S0020-0190(00)00196-4).
- [41] C. Flanagan and K. R. M. Leino, “Houdini, an annotation assistant for es-c/java”, in *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, J. N. Oliveira and P. Zave, Eds., ser. Lecture Notes in Computer Science, vol. 2021, Springer, 2001, pp. 500–517. DOI: [10.1007/3-540-45251-6_29](https://doi.org/10.1007/3-540-45251-6_29). [Online]. Available: https://doi.org/10.1007/3-540-45251-6%5C_29.
- [42] V. Ganesh, C. W. O’Donnell, M. Soos, S. Devadas, M. C. Rinard, and A. Solar-Lezama, “Lynx: A programmatic SAT solver for the rna-folding problem”, in *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, A. Cimatti and R.

- Sebastiani, Eds., ser. Lecture Notes in Computer Science, vol. 7317, Springer, 2012, pp. 143–156. DOI: [10.1007/978-3-642-31612-8_12](https://doi.org/10.1007/978-3-642-31612-8_12). [Online]. Available: https://doi.org/10.1007/978-3-642-31612-8_12.
- [43] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, “An abstract domain of uninterpreted functions”, in *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, B. Jobstmann and K. R. M. Leino, Eds., ser. Lecture Notes in Computer Science, vol. 9583, Springer, 2016, pp. 85–103. DOI: [10.1007/978-3-662-49122-5_4](https://doi.org/10.1007/978-3-662-49122-5_4).
 - [44] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “DPLL(T): fast decision procedures”, in *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, R. Alur and D. A. Peled, Eds., ser. Lecture Notes in Computer Science, vol. 3114, Springer, 2004, pp. 175–188. DOI: [10.1007/978-3-540-27813-9_14](https://doi.org/10.1007/978-3-540-27813-9_14). [Online]. Available: https://doi.org/10.1007/978-3-540-27813-9_14.
 - [45] I. Garcia-Contreras, H. G. V. K., S. Shoham, and A. Gurfinkel, “Fast approximations of quantifier elimination”, in *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, C. Enea and A. Lal, Eds., ser. Lecture Notes in Computer Science, vol. 13965, Springer, 2023, pp. 64–86. DOI: [10.1007/978-3-031-37703-7_4](https://doi.org/10.1007/978-3-031-37703-7_4). [Online]. Available: https://doi.org/10.1007/978-3-031-37703-7_4.
 - [46] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo, “Incremental and modular context-sensitive analysis”, *Theory Pract. Log. Program.*, vol. 21, no. 2, pp. 196–243, 2021. DOI: [10.1017/S1471068420000496](https://doi.org/10.1017/S1471068420000496). [Online]. Available: <https://doi.org/10.1017/S1471068420000496>.
 - [47] P. Garg, D. Neider, P. Madhusudan, and D. Roth, “Learning invariants using decision trees and implication counterexamples”, in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016, pp. 499–512. DOI: [10.1145/2837614.2837664](https://doi.org/10.1145/2837614.2837664). [Online]. Available: <https://doi.org/10.1145/2837614.2837664>.
 - [48] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanovic, and S. Malik, “Template-based circuit understanding”, in *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, IEEE, 2014, pp. 83–90. DOI: [10.1109/FMCAD.2014.6987599](https://doi.org/10.1109/FMCAD.2014.6987599).

- [49] A. V. Gelder, “Verifying RUP proofs of propositional unsatisfiability”, in *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008. [Online]. Available: http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008%5C_0008%5C_60a1f9b2fd607a61ec9e0feac3f438f8.pdf.
- [50] E. I. Goldberg and Y. Novikov, “Verification of proofs of unsatisfiability for CNF formulas”, in *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, IEEE Computer Society, 2003, pp. 10 886–10 891. DOI: [10.1109/DATE.2003.10008](https://doi.org/10.1109/DATE.2003.10008). [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/DATE.2003.10008>.
- [51] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, “Synthesizing software verifiers from proof rules”, in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds., ACM, 2012, pp. 405–416. DOI: [10.1145/2254064.2254112](https://doi.org/10.1145/2254064.2254112). [Online]. Available: <https://doi.org/10.1145/2254064.2254112>.
- [52] S. Gulwani and M. Musuvathi, “Cover algorithms and their combination”, in *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, S. Drossopoulou, Ed., ser. Lecture Notes in Computer Science, vol. 4960, Springer, 2008, pp. 193–207. DOI: [10.1007/978-3-540-78739-6_16](https://doi.org/10.1007/978-3-540-78739-6_16). [Online]. Available: https://doi.org/10.1007/978-3-540-78739-6_16.
- [53] S. Gulwani, A. Tiwari, and G. C. Necula, “Join algorithms for the theory of uninterpreted functions”, in *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*, K. Lodaya and M. Mahajan, Eds., ser. Lecture Notes in Computer Science, vol. 3328, Springer, 2004, pp. 311–323. DOI: [10.1007/978-3-540-30538-5_26](https://doi.org/10.1007/978-3-540-30538-5_26).
- [54] A. Gurfinkel and A. Ivrii, “Pushing to the top”, in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, R. Kaivola and T. Wahl, Eds., IEEE, 2015, pp. 65–72. DOI: [10.1109/FMCAD.2015.7542254](https://doi.org/10.1109/FMCAD.2015.7542254). [Online]. Available: <https://doi.org/10.1109/FMCAD.2015.7542254>.
- [55] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, “The seahorn verification framework”, in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings*,

- Part I*, D. Kroening and C. S. Pasareanu, Eds., ser. Lecture Notes in Computer Science, vol. 9206, Springer, 2015, pp. 343–361. DOI: [10.1007/978-3-319-21690-4_20](https://doi.org/10.1007/978-3-319-21690-4_20). [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4%5C_20.
- [56] A. Gurfinkel, P. Ruemmer, G. Fedyukovich, and A. Champion, *CHC-COMP*, <https://chc-comp.github.io/>, 2018.
 - [57] A. Gurfinkel, S. Shoham, and Y. Vizel, “Quantifiers on demand”, in *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, S. K. Lahiri and C. Wang, Eds., ser. Lecture Notes in Computer Science, vol. 11138, Springer, 2018, pp. 248–266. DOI: [10.1007/978-3-030-01090-4_15](https://doi.org/10.1007/978-3-030-01090-4_15). [Online]. Available: https://doi.org/10.1007/978-3-030-01090-4%5C_15.
 - [58] A. Gurfinkel and Y. Vizel, “DRUPing for interpolates”, in *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, IEEE, 2014, pp. 99–106. DOI: [10.1109/FMCAD.2014.6987601](https://doi.org/10.1109/FMCAD.2014.6987601). [Online]. Available: <https://doi.org/10.1109/FMCAD.2014.6987601>.
 - [59] M. Heule, W. A. H. Jr., and N. Wetzler, “Trimming while checking clausal proofs”, in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, IEEE, 2013, pp. 181–188. [Online]. Available: <https://ieeexplore.ieee.org/document/6679408/>.
 - [60] K. Hoder and N. Bjørner, “Generalized property directed reachability”, in *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, A. Cimatti and R. Sebastiani, Eds., ser. Lecture Notes in Computer Science, vol. 7317, Springer, 2012, pp. 157–171. DOI: [10.1007/978-3-642-31612-8_13](https://doi.org/10.1007/978-3-642-31612-8_13). [Online]. Available: https://doi.org/10.1007/978-3-642-31612-8%5C_13.
 - [61] H. Hojjat and P. Rümmer, “The ELDARICA horn solver”, in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, N. Bjørner and A. Gurfinkel, Eds., IEEE, 2018, pp. 1–7. DOI: [10.23919/FMCAD.2018.8603013](https://doi.org/10.23919/FMCAD.2018.8603013).
 - [62] S. Itzhaky, S. Shoham, and Y. Vizel, “Hyperproperty verification as CHC satisfiability”, in *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*, S. Weirich, Ed., ser. Lecture Notes in Computer Science, vol. 14577, Springer, 2024, pp. 212–241. DOI: [10.1007/978-3-031-57267-8_9](https://doi.org/10.1007/978-3-031-57267-8_9). [Online]. Available: https://doi.org/10.1007/978-3-031-57267-8%5C_9.

- [63] R. Joshi, G. Nelson, and K. H. Randall, “Denali: A goal-directed superoptimizer”, in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, J. Knoop and L. J. Hendren, Eds., ACM, 2002, pp. 304–314. DOI: [10.1145/512529.512566](https://doi.org/10.1145/512529.512566).
- [64] D. Jovanovic and B. Dutertre, “Property-directed k-induction”, in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, 2016, pp. 85–92. DOI: [10.1109/FMCAD.2016.7886665](https://doi.org/10.1109/FMCAD.2016.7886665). [Online]. Available: <https://doi.org/10.1109/FMCAD.2016.7886665>.
- [65] H. G. V. K., G. Fedyukovich, and A. Gurfinkel, “Word level property directed reachability”, in *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*, IEEE, 2020, 107:1–107:9. DOI: [10.1145/3400302.3415708](https://doi.org/10.1145/3400302.3415708). [Online]. Available: <https://doi.org/10.1145/3400302.3415708>.
- [66] H. G. V. K., I. Garcia-Contreras, S. Shoham, and A. Gurfinkel, “Speculative SAT modulo SAT”, in *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, B. Finkbeiner and L. Kovács, Eds., ser. Lecture Notes in Computer Science, vol. 14570, Springer, 2024, pp. 43–60. DOI: [10.1007/978-3-031-57246-3_4](https://doi.org/10.1007/978-3-031-57246-3_4). [Online]. Available: https://doi.org/10.1007/978-3-031-57246-3_4.
- [67] H. G. V. K., S. Shoham, and A. Gurfinkel, “Solving constrained horn clauses modulo algebraic data types and recursive functions”, *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–29, 2022. DOI: [10.1145/3498722](https://doi.org/10.1145/3498722). [Online]. Available: <https://doi.org/10.1145/3498722>.
- [68] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäfer, “Jayhorn: A framework for verifying java programs”, in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, S. Chaudhuri and A. Farzan, Eds., ser. Lecture Notes in Computer Science, vol. 9779, Springer, 2016, pp. 352–358. DOI: [10.1007/978-3-319-41528-4_19](https://doi.org/10.1007/978-3-319-41528-4_19). [Online]. Available: https://doi.org/10.1007/978-3-319-41528-4_19.
- [69] J. Kim, Q. Hu, L. D’Antoni, and T. W. Reps, “Semantics-guided synthesis”, *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–32, 2021. DOI: [10.1145/3434311](https://doi.org/10.1145/3434311). [Online]. Available: <https://doi.org/10.1145/3434311>.

- [70] A. Komuravelli, N. S. Bjørner, A. Gurfinkel, and K. L. McMillan, “Compositional verification of procedural programs using horn clauses over integers and arrays”, in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, R. Kaivola and T. Wahl, Eds., IEEE, 2015, pp. 89–96. DOI: [10.5555/2893529.2893548](https://doi.org/10.5555/2893529.2893548).
- [71] A. Komuravelli, A. Gurfinkel, and S. Chaki, “SMT-based model checking for recursive programs”, in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, 2014, pp. 17–34. DOI: [10.1007/978-3-319-08867-9_2](https://doi.org/10.1007/978-3-319-08867-9_2). [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_2.
- [72] H. G. V. Krishnan, Y. Chen, S. Shoham, and A. Gurfinkel, “Global guidance for local generalization in model checking”, in *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, S. K. Lahiri and C. Wang, Eds., ser. Lecture Notes in Computer Science, vol. 12225, Springer, 2020, pp. 101–125. DOI: [10.1007/978-3-030-53291-8_7](https://doi.org/10.1007/978-3-030-53291-8_7). [Online]. Available: https://doi.org/10.1007/978-3-030-53291-8_7.
- [73] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, “Complete functional synthesis”, in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, B. G. Zorn and A. Aiken, Eds., ACM, 2010, pp. 316–329. DOI: [10.1145/1806596.1806632](https://doi.org/10.1145/1806596.1806632).
- [74] N. Le, X. Si, and A. Gurfinkel, “Data-driven optimization of inductive generalization”, in *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*, IEEE, 2021, pp. 86–95. DOI: [10.34727/2021/ISBN.978-3-85448-046-4_17](https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4_17). [Online]. Available: https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4_17.
- [75] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness”, in *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, E. M. Clarke and A. Voronkov, Eds., ser. Lecture Notes in Computer Science, vol. 6355, Springer, 2010, pp. 348–370. DOI: [10.1007/978-3-642-17511-4_20](https://doi.org/10.1007/978-3-642-17511-4_20). [Online]. Available: https://doi.org/10.1007/978-3-642-17511-4_20.
- [76] K. R. M. Leino and V. Wüstholtz, “The dafny integrated development environment”, in *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014*, C. Dubois, D.

- Giannakopoulou, and D. Méry, Eds., ser. EPTCS, vol. 149, 2014, pp. 3–15. DOI: [10.4204/EPTCS.149.2](https://doi.org/10.4204/EPTCS.149.2). [Online]. Available: <https://doi.org/10.4204/EPTCS.149.2>.
- [77] J. Leroux, P. Rümmer, and P. Subotic, “Guiding Craig interpolation with domain-specific abstractions”, *Acta Inf.*, vol. 53, no. 4, pp. 387–424, 2016. DOI: [10.1007/s00236-015-0236-z](https://doi.org/10.1007/s00236-015-0236-z). [Online]. Available: <https://doi.org/10.1007/s00236-015-0236-z>.
 - [78] C. Löding, P. Madhusudan, and L. Peña, “Foundations for natural proofs and quantifier instantiation”, *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 10:1–10:30, 2018. DOI: [10.1145/3158098](https://doi.org/10.1145/3158098).
 - [79] Y. Matsushita, T. Tsukada, and N. Kobayashi, “Rusthorn: Chc-based verification for rust programs”, in *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, P. Müller, Ed., ser. Lecture Notes in Computer Science, vol. 12075, Springer, 2020, pp. 484–514. DOI: [10.1007/978-3-030-44914-8_18](https://doi.org/10.1007/978-3-030-44914-8_18).
 - [80] K. L. McMillan, “Interpolation and SAT-Based Model Checking”, in *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, 2003, pp. 1–13. DOI: [10.1007/978-3-540-45069-6_1](https://doi.org/10.1007/978-3-540-45069-6_1). [Online]. Available: https://doi.org/10.1007/978-3-540-45069-6_1.
 - [81] K. L. McMillan, “Lazy abstraction with interpolants”, in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, 2006, pp. 123–136. DOI: [10.1007/11817963_14](https://doi.org/10.1007/11817963_14). [Online]. Available: https://doi.org/10.1007/11817963_14.
 - [82] K. L. McMillan, A. Kuehlmann, and M. Sagiv, “Generalizing DPLL to richer logics”, in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, 2009, pp. 462–476. DOI: [10.1007/978-3-642-02658-4_35](https://doi.org/10.1007/978-3-642-02658-4_35). [Online]. Available: https://doi.org/10.1007/978-3-642-02658-4_35.
 - [83] K. L. McMillan and O. Padon, “Ivy: A multi-modal verification tool for distributed algorithms”, in *Computer Aided Verification - 32nd International Conference, CAV 2020, , Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, S. K. Lahiri and C. Wang, Eds., ser. Lecture Notes in Computer Science, vol. 12225, Springer, 2020, pp. 190–202. DOI: [10.1007/978-3-030-53291-8_12](https://doi.org/10.1007/978-3-030-53291-8_12). [Online]. Available: https://doi.org/10.1007/978-3-030-53291-8_12.

- [84] M. Méndez-Lojo, J. A. Navas, and M. V. Hermenegildo, “A flexible, (c)lp-based approach to the analysis of object-oriented programs”, in *Logic-Based Program Synthesis and Transformation, 17th International Symposium, LOPSTR 2007, Kongens Lyngby, Denmark, August 23-24, 2007, Revised Selected Papers*, A. King, Ed., ser. Lecture Notes in Computer Science, vol. 4915, Springer, 2007, pp. 154–168. DOI: [10.1007/978-3-540-78769-3_11](https://doi.org/10.1007/978-3-540-78769-3_11). [Online]. Available: https://doi.org/10.1007/978-3-540-78769-3_11.
- [85] L. M. de Moura and N. S. Bjørner, “Z3: an efficient SMT solver”, in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, C. R. Ramakrishnan and J. Rehof, Eds., ser. Lecture Notes in Computer Science, vol. 4963, Springer, 2008, pp. 337–340. DOI: [10.1007/978-3-540-78800-3_24](https://doi.org/10.1007/978-3-540-78800-3_24). [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24.
- [86] P. Müller, M. Schwerhoff, and A. J. Summers, “Viper: A verification infrastructure for permission-based reasoning”, in *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, B. Jobstmann and K. R. M. Leino, Eds., ser. Lecture Notes in Computer Science, vol. 9583, Springer, 2016, pp. 41–62. DOI: [10.1007/978-3-662-49122-5_2](https://doi.org/10.1007/978-3-662-49122-5_2). [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5_2.
- [87] S. Nejati, J. H. Liang, V. Ganesh, C. Gebotys, and K. Czarnecki, “Sha-1 preimage instances for sat”, *SAT COMPETITION 2017*, p. 45,
- [88] G. Nelson and D. C. Oppen, “Fast decision algorithms based on union and find”, in *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, IEEE Computer Society, 1977, pp. 114–119. DOI: [10.1109/SFCS.1977.12](https://doi.org/10.1109/SFCS.1977.12).
- [89] G. Nelson and D. C. Oppen, “Simplification by cooperating decision procedures”, *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 2, pp. 245–257, 1979. DOI: [10.1145/357073.357079](https://doi.org/10.1145/357073.357079).
- [90] G. PLOTKIN, “A note on inductive generalization”, *Machine Intelligence*, vol. 5, pp. 153–163, 1969.
- [91] M. Research. “Project Everest”. (1999), [Online]. Available: <https://project-everest.github.io/> (visited on 07/31/2021).

- [92] A. Reynolds and V. Kuncak, “Induction for SMT solvers”, in *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, D. D’Souza, A. Lal, and K. G. Larsen, Eds., ser. Lecture Notes in Computer Science, vol. 8931, Springer, 2015, pp. 80–98. DOI: [10.1007/978-3-662-46081-8_5](https://doi.org/10.1007/978-3-662-46081-8_5).
- [93] P. M. Rondon, M. Kawaguchi, and R. Jhala, “Liquid types”, in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, R. Gupta and S. P. Amarasinghe, Eds., ACM, 2008, pp. 159–169. DOI: [10.1145/1375581.1375602](https://doi.org/10.1145/1375581.1375602). [Online]. Available: <https://doi.org/10.1145/1375581.1375602>.
- [94] D. Rosén and N. Smallbone, “TIP: tools for inductive provers”, in *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, M. Davis, A. Fehnker, A. McIver, and A. Voronkov, Eds., ser. Lecture Notes in Computer Science, vol. 9450, Springer, 2015, pp. 219–232. DOI: [10.1007/978-3-662-48899-7_16](https://doi.org/10.1007/978-3-662-48899-7_16). [Online]. Available: https://doi.org/10.1007/978-3-662-48899-7_16.
- [95] T. Schindler and D. Jovanovic, “Selfless interpolation for infinite-state model checking”, in *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, 2018, pp. 495–515. DOI: [10.1007/978-3-319-73721-8_23](https://doi.org/10.1007/978-3-319-73721-8_23). [Online]. Available: https://doi.org/10.1007/978-3-319-73721-8_23.
- [96] P. Suter, M. Dotta, and V. Kuncak, “Decision procedures for algebraic data types with abstractions”, in *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, M. V. Hermenegildo and J. Palsberg, Eds., ACM, 2010, pp. 199–210. DOI: [10.1145/1706299.1706325](https://doi.org/10.1145/1706299.1706325).
- [97] P. Suter, A. S. Köksal, and V. Kuncak, “Satisfiability modulo recursive programs”, in *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, E. Yahav, Ed., ser. Lecture Notes in Computer Science, vol. 6887, Springer, 2011, pp. 298–315. DOI: [10.1007/978-3-642-23702-7_23](https://doi.org/10.1007/978-3-642-23702-7_23).
- [98] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, “Equality saturation: A new approach to optimization”, in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’09, Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 264–276, ISBN: 9781605583792. DOI: [10.1145/1480881.1480915](https://doi.org/10.1145/1480881.1480915).

- [99] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, “Equality saturation: A new approach to optimization”, *Log. Methods Comput. Sci.*, vol. 7, no. 1, 2011. DOI: [10.2168/LMCS-7\(1:10\)2011](https://doi.org/10.2168/LMCS-7(1:10)2011).
- [100] H. G. Veditramana Krishnan, Y. Chen, S. Shoham, and A. Gurfinkel, “Global guidance for local generalization in model checking”, *Formal Methods in System Design*, Mar. 2023, ISSN: 1572-8102. DOI: [10.1007/s10703-023-00412-3](https://doi.org/10.1007/s10703-023-00412-3). [Online]. Available: <https://doi.org/10.1007/s10703-023-00412-3>.
- [101] Y. Vizel and A. Gurfinkel, “Interpolating property directed reachability”, in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, A. Biere and R. Bloem, Eds., ser. Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 260–276. DOI: [10.1007/978-3-319-08867-9_17](https://doi.org/10.1007/978-3-319-08867-9_17). [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_17.
- [102] T. Welp and A. Kuehlmann, “QF_BV model checking with property directed reachability”, in *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, 2013, pp. 791–796. DOI: [10.7873/DATE.2013.168](https://doi.org/10.7873/DATE.2013.168). [Online]. Available: <https://doi.org/10.7873/DATE.2013.168>.
- [103] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, “Egg: Fast and extensible equality saturation”, *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–29, 2021. DOI: [10.1145/3434304](https://doi.org/10.1145/3434304).
- [104] G. Yernaux and W. Vanhoof, “Anti-unification in constraint logic programming”, *TPLP*, vol. 19, no. 5-6, pp. 773–789, 2019. DOI: [10.1017/S1471068419000188](https://doi.org/10.1017/S1471068419000188). [Online]. Available: <https://doi.org/10.1017/S1471068419000188>.
- [105] T. Zhang, H. B. Sipma, and Z. Manna, “Decision procedures for recursive data structures with integer constraints”, in *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, D. A. Basin and M. Rusinowitch, Eds., ser. Lecture Notes in Computer Science, vol. 3097, Springer, 2004, pp. 152–167. DOI: [10.1007/978-3-540-25984-8_9](https://doi.org/10.1007/978-3-540-25984-8_9). [Online]. Available: https://doi.org/10.1007/978-3-540-25984-8_9.
- [106] H. Zhu, S. Magill, and S. Jagannathan, “A data-driven CHC solver”, in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds., ACM, 2018, pp. 707–721. DOI: [10.1145/3192366.3192416](https://doi.org/10.1145/3192366.3192416). [Online]. Available: <https://doi.org/10.1145/3192366.3192416>.