TU WIEN Informatics

Inductive Reasoning in Superposition

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Technischen Wissenschaften

by

Mgr. Petra Hozzová
Registration Number 11934931

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Laura Kovács
Second advisor: Prof. Dr. Andrei Voronkov

The dissertation has been reviewed by:

Jasmin Blanchette	Viorica Sofronie-Stokkermans

Vienna, June 3, 2024

Petra Hozzová

Erklärung zur Verfassung der Arbeit

Mgr. Petra Hozzová

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

Wien, 3. Juni 2024

Petra Hozzová

Acknowledgements

This PhD was quite a ride! And as any great adventure, it was enabled by many people to whom I am deeply grateful.

First and foremost, I would like to express my gratitude to my advisor Laura, whose guidance in all areas was integral to my journey. Further, I also thank my second advisor Andrei for many valuable discussions. Moreover, I would like to thank the reviewers of this thesis Jasmin Blanchette and Viorica Sofronie-Stokkerman, whose feedback improved the final product and provided me with food for future thought. Additionally, to my coauthors Márton, Joe, Chase, Daneshvar, Eva, Jakob, Michael, Martin, Giles, Jarda, Alex, and Yoav: thank you, it was a pleasure to work with you – and I hope to have an opportunity to do so again in the future!

My heartfelt thanks go to my colleagues, who along the way became my friends: Anton, Clemens, Friedrich, Liana, Márton, Pamina, and Sarah. The whole PhD experience managed to surprise me in many ways, but finding such great people was one of the nicest surprises of all. Дзякуй, danke, köszönöm!

I would also like to extend my gratitude to my closest friends from my non-academic life. Moni, Máška, Mário, Žaba, thank you for keeping me going and keeping me in check. Finally, a huge thank you to my family: my parents Zuzka and Palko, whose unwavering support helped me stay on my path, and my amazing husband Janko, who was my point of stability throughout all the ups and downs. Ďakujem!

Kurzfassung

In dieser Arbeit erweiteren wir automatisiertes Beweisen für formale Verifikation und Programmsynthese mit Induktion. Wir arbeiten mit Systemen, die auf der Sättigung einer initialen Klauselmenge durch logische Schlüsse basieren, im Weiteren *sättigungsbasierte Systeme* gennant. Der Fokus liegt dabei auf Theorembeweisern für Prädikatenlogik erster Stufe, die wir *Beweiser erster Stufe* nennen.

Theorien von induktiv definierte Datentypen wie natürliche Zahlen oder Listen sowie die Theorie der Ganzzahlarithmetik werden häufig bei der Entwicklung imperativer und funktionaler Programme verwendet. Darüber hinaus erfordern Beweise über Schleifen oder Rekursion häufig das Induktionsprinzip. Daher müssen automatische Beweistechniken für die formale Verifikation von Programmen auch Induktion für die Typen der oben genannten Theorien automatisieren. Um der Forderung nach vertrauenswürdigen Softwaresystemen angemessen gerecht zu werden, konzentriert sich diese Arbeit auf (1) die Automatisierung von induktiven Schlüssen in Therembeweisern für der Prädikatenlogik erster Stufe und (2) die Synthese von Programmen auf der Grundlage von Beweisen in Prädikatenlogik erster Stufe, die möglicherweise Induktion verwenden.

Im ersten Teil der Arbeit erweitern wir induktive Beweisverfahren für sättigungsbasierte Theorembeweiser erster Stufe. Die Herausforderung besteht darin, dass sich das Framework erheblich von den meisten induktiven Beweisern unterscheidet – es unterstützt nicht die Ziel-/Unterzielarchitektur. Wir integrieren daher Induktion in das Superpositionskalkül, das von sättigungsbasierte Systeme verwendet wird, und verwenden weder Termersetzungsregeln noch externe Heuristiken für die Erzeugung zusätzlicher induktiver Lemmas.

Im zweiten Teil schlagen wir ein deduktives Programmsynthese-Framework vor, das auf dem sättigungsbasierten System basiert. Wir verwenden Theorembeweisen als Grundlage für die Synthese ausgehend von einer Funktionsspezifikation, die als Formel in Prädikatenlogik erster Stufe gegeben ist und die Existenz eines bestimmten Programms ausdrückt. Beim Beweisen der Existenz eines Programms synthetisieren wir auch das Programm, das per Konstruktion korrekt ist. Wir beginnen mit der Konstruktion von rekursionsfreien Programmen aus induktionsfreien Beweisen und erweitern diesen Ansatz dann, um auch einfache rekursive Programme aus Beweisen mit Induktion zu synthetisieren.

Wir haben die in dieser Arbeit beschriebenen induktiven Beweistechniken im sättigungsbasierten Beweiser Vampire implementiert. Wir präsentieren eine Reihe von experimentellen Auswertungen unserer Implementierung und zeigen, dass die in dieser Arbeit vorgeschlagenen Ansätze in der Praxis gut funktionieren.

Abstract

In this thesis, we focus on extending automated reasoning for formal verification with induction.

Theories of inductively defined data types, such as natural numbers or lists, and the theory of integer arithmetic are commonly used in the development of imperative and functional programs. Further, reasoning about loops or recursion often requires the inductive principle. Therefore, automating reasoning in formal verification of programs also needs to automate induction over the types of the aforementioned theories. To adequately respond to the demand of ensuring trustworthiness of software systems, this thesis focuses on (1) mechanizing inductive reasoning within first-order theorem proving, and (2) constructing programs based on first-order proofs possibly using induction.

In the first part, we extend the inductive reasoning capabilities for the saturation-based framework of automated first-order theorem provers. The challenge is that the framework substantially differs from most inductive provers – it does not support the goal/subgoal architecture. We therefore integrate induction with the superposition calculus used by the saturation framework, and do not use rewrite rules nor external heuristics for generating auxiliary inductive lemmas.

In the second part, we propose a deductive program synthesis framework based on saturation. We use theorem proving as a basis for synthesis from a functional specification given as a first-order formula expressing the existence of a particular program. In the process of proving the existence of a program, we also synthesize the program, which is correct by construction. We begin with constructing recursion-free programs from induction-free proofs, and then we extend this approach to also synthesize simple recursive programs from proofs using induction.

We implemented the inductive reasoning techniques described in this thesis in the saturation-based theorem prover Vampire. We present a set of experimental evaluations of our implementation, demonstrating that the approaches proposed in this thesis work well in practice.

Contents

CHAPTER 1

Introduction

As our world undergoes transformation to information society, more and more systems are controlled by software. This includes safety-critical systems and systems with access to valuable assets or sensitive data. The incentives for making all this software reliable are therefore high. However, verifying correctness of software is a hard problem. Testing is a straightforward method for discovering bugs, but short of checking the software behavior on all possible inputs it cannot guarantee that the software will always behave as it should. While there is no universal technique for certifying software correctness, the methods of *formal verification* can do so in certain cases.

In a nutshell, formal verification works as follows. To verify that a program is indeed correct, we first formulate the correctness specification. Then we encode both the program and the specification in some suitable formal language(s), obtaining the encodings P and S, respectively, for the program and its specification. Finally, we check whether P satisfies S – if yes, this gives us some (possibly limited) guarantee of the program correctness with respect to the specification. All three steps of this recipe for program verification require some heavy lifting: in many cases it is not easy to formulate what exactly it means for a program to be correct; depending on the program and the specification, a rigorous and efficient encoding in the target formal language might not even exist; and finally, for many formal languages that are expressive enough to encode the program and specification as P and S, checking whether P satisfies S is not decidable. The third challenge is closely related to the problem of proving mathematical theorems, for which there is also no universal algorithm. However, recent developments in automated reasoning open up new avenues to tackling the third challenge by automatically checking whether P satisfies S.

In this thesis we advocate the use of *automated theorem proving for facilitating program verification and synthesis*. We assume that the program and the specification are encoded as formulas P and S, respectively, in first-order logic with theories, a language rich enough to model most standard constructions. Then we apply the methods of fully automated first-order theorem proving to formally prove that P satisfies S. The focus of this thesis is on *development of the reasoning methods* with the aim of categorically expanding the set of formulas for which we can prove that P satisfies S.

A feature crucial for reasoning about programs is *induction*: this is the principle that allows for reasoning about loop iterations or recursive calls. If we can prove that a property (i) holds at the beginning of the loop, and that (ii) it is an invariant of the loop (i.e., if it holds after the ith iteration, it will also hold after the $i+1$st iteration), then by the induction principle the property holds after any iteration of the loop – including the last one, if the loop terminates. Similarly, consider a recursive function definition consisting of a base case and a recursive case. Assume that we can prove that a property (i) holds for the value returned in the base case, and that (ii) the property holding for the value returned in the recursive case follows from the property holding for the values returned by any recursive call within the recursive case. Then, by the induction principle the property holds for the value computed by the function when applied to any argument.

However, until recently *fully automated inductive reasoning*[1] has been the domain of inductive theorem provers [BM79, BSvH$^+$93, CJRS12, SDE12, PCI$^+$20], which lack other capabilities necessary for reasoning about programs, such as efficient reasoning about quantifiers or theories. On the other hand, theorem provers and SMT solvers, which traditionally focused on the quantified and theory-specific areas of the automated reasoning landscape, respectively, did not support induction. This changed recently with the advances in fully automated reasoning with first-order logic and SMT related to induction, such as first-order reasoning with inductively defined data types [KRV17], the Avatar architecture [Vor14], inductive strengthening of SMT properties [RK15], structural induction in superposition [Cru17] and general induction rules within saturation [RV19]. These advances make it possible to re-consider the grand challenge of mechanizing mathematical induction [BM79] in the context of reasoning with full first-order logic with theories.

1.1 Contributions

In our work, we focus on *saturation-based first-order theorem proving using the superposition calculus*. This is the leading approach to theorem proving, as evidenced by the automated theorem proving competition CASC [Sut16], winners of which predominantly use this paradigm. The work of [RV19] naturally extended the superposition calculus in saturation with an induction rule. This thesis consists of two parts, both of which are based on and further develop superposition in saturation extended with induction.

In the first part of this thesis, we *extend the induction rule in superposition* by introducing new axioms to instantiate the rule with. Our new axioms allow us to fully automatically solve problems coming from verification, as well as mathematics, that were not automatically solvable before. This includes two types of properties. First, properties that are not straightforwardly provable despite being instances of other inductive properties,

[1]In this thesis we use the term *inductive reasoning* with the meaning “reasoning with induction”, not “generalizing from examples”.

which can themselves be proved easily. The challenge our work addresses is automating the discovery of a generalization of the original property, and the subsequent direct use of the induction axiom for the generalization for proving the original property. Second, we investigate proving of inductive properties over integers. Unlike inductively defined data types, the set of integers with the standard less-than order is not well-founded, and thus there is also no specific value that would be a natural candidate for a base case of an induction axiom. However, we can apply inductive reasoning over any interval of integers where at least one of the bounds is not (negative) infinity. Here, the automation challenge we solve is how to choose the bounds, and thus also the base case and induction step for the induction axiom. Finally, we also present a benchmark set we created while developing the methods mentioned in this paragraph, since previously there were no benchmarks focused on induction with generalization and integer induction.

In the second part of this thesis, we turn the verification problem upside-down and use saturation-based proving to *automate program synthesis*. Instead of taking a program and a specification and proving that the program satisfies the specification, we only consider the specification and automatically synthesize a program satisfying it. We focus on functional specifications summarized by valid first-order formulas expressing the existence of a program computing the desired output for a given input [MW80, ABD$^+$15]. While being a powerful alternative to formal verification [SGF10], program synthesis faces intrinsic computational challenges. One of these challenges is posed to the reasoning backend used for handling program specifications, as the latter typically include first-order quantifier alternations and interpreted theory symbols. As such, efficient reasoning with both theories and quantifiers is imperative for any effort toward program synthesis. We address this challenge by integrating synthesis into saturation-based proof search, thereby obtaining a *saturation-based synthesis algorithm*. The main idea is that we can construct the program in parallel with proving its existence. We obtain fragments of the sought program from substitutions used during the proof. Further, we construct the program structure based on the proof structure. We translate branching from the proof (e.g. when a resolution rule is used) to branching in the program using the `if−then−else` construction. Finally, we take advantage of all the developments of inductive proving, and utilize the connection between induction and recursion to translate induction from the proof into recursion in the constructed program. Briefly, we construct the base case for a recursive function in parallel with proving the base case of an induction axiom, and the recursive case when proving the induction step of the induction axiom. As a result, we fully automatically synthesize programs using primitive recursion.

We implemented all our results in the superposition-based theorem prover Vampire [KV13]. Our experimental evaluation shows that our methods are also practically viable and allow us to solve many problems originating from the areas of program analysis and mathematics that were not automatically solvable before.

1.2 Publications and Relation to Contributions

The contributions of this thesis are based on the following peer-reviewed publications, and one workshop proceedings contribution, for which *I acted as the main author*:

[HHK$^+$20] Márton Hajdu, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Induction with Generalization in Superposition Reasoning. In Christoph Benzmüller and Bruce Miller, editors, *Proc. of CICM*, volume 12236 of *LNCS*, pages 123–137, Cham, 2020. Springer
Chapter 3 and Section 6.1, as well as parts of Chapter 5 are based on this peer-reviewed publication.

[HKV21] Petra Hozzová, Laura Kovács, and Andrei Voronkov. Integer Induction in Saturation. In André Platzer and Geoff Sutcliffe, editors, *Proc. of CADE*, volume 12699 of *LNCS*, pages 361–377, Cham, 2021. Springer
Chapter 4 and Section 6.2, as well as parts of Chapter 5 are based on this peer-reviewed publication.

[HHK$^+$21] Márton Hajdu, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Inductive Benchmarks for Automated Reasoning. In Fairouz Kamareddine and Claudio Sacerdoti Coen, editors, *Proc. of CICM*, volume 12833 of *LNCS*, pages 124–129, Cham, 2021. Springer
The main content of Chapter 5 is based on this peer-reviewed publication.

[HKNV23] Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov. Program Synthesis in Saturation. In Brigitte Pientka and Cesare Tinelli, editors, *Proc. of CADE*, volume 14132 of *LNCS*, pages 307–324, Cham, 2023. Springer
Chapter 7 and Section 9.1 are based on this peer-reviewed publication, which received an honorable mention for best student paper at CADE 2023.

[Hoz24] Petra Hozzová. Integrating Answer Literals with AVATAR for Program Synthesis. In Laura Kovács and Michael Rawson, editors, *Proc. of the 7th and 8th Vampire Workshop*, volume 99 of *EPiC Series in Computing*, pages 13–20. EasyChair, 2024
Section 7.6 is based on this workshop proceedings contribution.

[HAH$^+$24] Petra Hozzová, Daneshvar Amrollahi, Márton Hajdu, Laura Kovács, Andrei Voronkov, and Eva Maria Wagner. Synthesis of Recursive Programs in Saturation. EasyChair Preprint no. 12145, EasyChair, 2024, to appear in *Proc. of IJCAR*, 2024
Chapter 8 and Section 9.2 are based on this peer-reviewed publication.

Additionally, in the course of my PhD I also co-authored the following publications:

[HKH$^+$20] Martin Homola, Ján Kľuka, Petra Hozzová, Vojtěch Svátek, and Miroslav Vacura. Towards Higher-Order OWL. *KI-Künstliche Intelligenz*, 34(3):417–421, 2020
I was a co-author of this project report not included in this thesis.

[HKR21] Petra Hozzová, Laura Kovács, and Jakob Rath. Automated Generation of Exam Sheets for Automated Deduction. In Fairouz Kamareddine and Claudio Sacerdoti Coen, editors, *Proc. of CICM*, volume 12833 of *Lecture Notes in Computer Science*, pages 185–196, Cham, 2021. Springer
I was a co-author of this publication not included in this thesis.

[HHKV21] Márton Hajdu, Petra Hozzová, Laura Kovács, and Andrei Voronkov. Induction with Recursive Definitions in Superposition. In Ruzica Piskac and Michael W. Whalen, editors, *Proc. of FMCAD*, pages 246–255. TU Wien Academic Press, 2021
I was a co-author of this publication not included in this thesis.

[RSHR22] Michael Rawson, Martin Suda, Petra Hozzová, and Giles Reger. Reuse of Introduced Symbols in Automatic Theorem Provers. In Boris Konev, Claudia Schon, and Alexander Steen, editors, *Proc. of PAAR*, volume 3201 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022
I was a co-author of this publication not included in this thesis.

[HHK$^+$22] Márton Hajdu, Petra Hozzová, Laura Kovács, Giles Reger, and Andrei Voronkov. *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, volume 13660 of *LNCS*, chapter Getting Saturated with Induction, pages 306–322. Springer, Cham, 2022
I was a co-author of this publication, which surveys (besides other work also) the contributions from [HHK$^+$20] and [HKV21], and therefore also covers parts of Chapters 3 and 4.

[HBNR23] Petra Hozzová, Jaroslav Bendík, Alexander Nutz, and Yoav Rodeh. Overapproximation of Non-Linear Integer Arithmetic for Smart Contract Verification. In Ruzica Piskac and Andrei Voronkov, editors, *Proc. of LPAR*, volume 94 of *EPiC Series in Computing*, pages 257–269. EasyChair, 2023
I was the main author of this publication not included in this thesis.

1.3 Outline

This thesis is organized as follows.

We open the thesis with preliminaries summarizing the background of saturation-based reasoning with superposition, induction, and synthesis in Chapter 2.

The first part of the thesis extends inductive proving in saturation. In Chapter 3 we describe our extension of induction by generalization [HHK$^+$20]. Then, in Chapter 4 we present our work on integer induction [HKV21]. Chapter 5 describes our benchmark set [HHK$^+$21] developed to evaluate our aforementioned work, and Chapter 6 reports on the implementation and evaluation of our inductive proving efforts.

The second part of the thesis introduces program synthesis in saturation. In Chapter 7 we present our program synthesis framework for the case of recursion-free programs [HKNV23, Hoz24]. Further, in Chapter 8 we describe an extension of our framework to recursive program synthesis [HAH$^+$24]. Then, in Chapter 9, we overview the implementation of our synthesis framework and survey examples and experimental results.

Finally, we review related work in Chapter 10 before concluding in Chapter 11.

CHAPTER 2

Preliminaries

We consider multi-sorted first-order logic (FOL) with equality. We allow all the standard logical connectives and quantifiers in the language: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \forall, \exists$. Additionally, we assume the logical constants $\top$ for true and $\bot$ for false, of the *boolean sort* Bool. Throughout this thesis we denote variables by $x, y, z, v, w, e, j, n, m$, constants by c, skolem constants by σ, terms by t, r, s, u, atoms by A, literals by L, formulas by F, G and clauses (disjunctions of literals) by C, D, all possibly with indices. We denote the equality predicate by $\simeq$ and write $t_1 \not\simeq t_2$ as a shorthand for $\neg(t_1 \simeq t_2)$. We reserve the symbol $\square$ for the *empty clause* which is logically equivalent to $\bot$. We write $\overline{L}$ for the literal complementary to L. We include a conditional term constructor `if`$-$`then`$-$`else` in the language, as follows: given a formula F and terms s, t of the same sort, we write `if` F `then` s `else` t to denote the term s if F is true and t otherwise.

An *expression* is a term, literal, clause, or formula. We write $E[t]$ to denote that the expression E contains the term t. For simplicity, $E[s]$ denotes the expression E where all occurrences of t are replaced by the term s. A *substitution* θ is a mapping from variables to terms. A substitution θ is a *unifier* of two expressions E and E' if $E\theta = E'\theta$, and is a *most general unifier* (*mgu*) if for every unifier η of E and E', there exists a substitution μ such that $\eta = \theta\mu$. We denote the mgu of E and E' with $\mathtt{mgu}(E, E')$.

A *universal closure* of a formula F is the formula $\forall\overline{z}.F$, where $\overline{z}$ are all free variables of F. We write $\mathtt{cnf}(F)$ and $\mathtt{cnf}(S)$ to mean (an arbitrary but fixed) clausal normal form (CNF) of a formula F and a set of formulas S, respectively. We consider free variables in clauses to be implicitly universally quantified.

We use a set of closed formulas defining a *theory* T. We consider T arbitrarily fixed and give all notions relative to T. For simplicity, we may drop the explicit reference to T. Symbols occurring in a theory T are *interpreted* and all other symbols are *uninterpreted*.

We work with *term algebras* [RV01], in particular with the special classes of the algebraically defined data types of the *natural numbers* $\mathbb{N}$, *lists* $\mathbb{L}$, and *binary trees* $\mathbb{BT}$.

Natural numbers $\mathbb{N}$	Constructors:	$\mathsf{0} : \mathbb{N}, \qquad \mathsf{s} : \mathbb{N} \to \mathbb{N}$
	Symbols:	$+_{\mathbb{N}} : \mathbb{N}\times\mathbb{N} \to \mathbb{N}, \qquad \cdot_{\mathbb{N}} : \mathbb{N}\times\mathbb{N} \to \mathrm{Bool}$
		$\mathsf{half} : \mathbb{N} \to \mathbb{N}, \qquad \leq_{\mathbb{N}} : \mathbb{N}\times\mathbb{N} \to \mathrm{Bool}$
	Axioms:	$\forall y \in \mathbb{N}.\ \mathsf{0} +_{\mathbb{N}} y \simeq y$
		$\forall x, y \in \mathbb{N}.\ \mathsf{s}(x) +_{\mathbb{N}} y \simeq \mathsf{s}(x +_{\mathbb{N}} y)$
		$\forall x \in \mathbb{N}.\ \mathsf{0} \cdot_{\mathbb{N}} x \simeq \mathsf{0}$
		$\forall x, y \in \mathbb{N}.\ \mathsf{s}(x) \cdot_{\mathbb{N}} y \simeq (x \cdot_{\mathbb{N}} y) + y$
		$\mathsf{half}(\mathsf{0}) \simeq \mathsf{0}$
		$\mathsf{half}(\mathsf{s}(\mathsf{0})) \simeq \mathsf{0}$
		$\forall x \in \mathbb{N}.\ \mathsf{half}(\mathsf{s}(\mathsf{s}(x))) \simeq \mathsf{s}(\mathsf{half}(x))$
		$\forall x \in \mathbb{N}.\ \mathsf{0} \leq_{\mathbb{N}} x$
		$\forall x \in \mathbb{N}.\ \neg \mathsf{s}(x) \leq_{\mathbb{N}} \mathsf{0}$
		$\forall x, y \in \mathbb{N}.\ (\mathsf{s}(x) \leq_{\mathbb{N}} \mathsf{s}(y) \leftrightarrow x \leq_{\mathbb{N}} y)$
Lists $\mathbb{L}$	Constructors:	$\mathsf{nil} : \mathbb{L}, \qquad \mathsf{cons} : \mathbb{N} \times \mathbb{L} \to \mathbb{L}$
	Symbols:	$\mathbin{+\!\!+} : \mathbb{L} \times \mathbb{L} \to \mathbb{L}, \qquad \mathsf{len} : \mathbb{L} \to \mathbb{N}, \qquad \mathsf{in}_{\mathbb{L}} : \mathbb{N} \times \mathbb{L} \to \mathrm{Bool},$
		$\mathsf{pref} : \mathbb{L} \times \mathbb{L} \to \mathrm{Bool}, \qquad \mathsf{suff} : \mathbb{L} \times \mathbb{L} \to \mathrm{Bool}$
	Axioms:	$\forall l \in \mathbb{L}.\ \mathsf{nil} \mathbin{+\!\!+} l \simeq l$
		$\forall x \in \mathbb{N}.\ \forall l, k \in \mathbb{L}.\ \mathsf{cons}(x, l) \mathbin{+\!\!+} k \simeq \mathsf{cons}(x, l \mathbin{+\!\!+} k)$
		$\mathsf{len}(\mathsf{nil}) \simeq \mathsf{0}$
		$\forall x \in \mathbb{N}.\ \forall l \in \mathbb{L}.\ \mathsf{len}(\mathsf{cons}(x, l)) \simeq \mathsf{s}(\mathsf{len}(l))$
		$\forall l \in \mathbb{L}.\ \mathsf{pref}(\mathsf{nil}, l)$
		$\forall x \in \mathbb{N}.\ \forall l \in \mathbb{L}.\ \neg \mathsf{pref}(\mathsf{cons}(x, l), \mathsf{nil})$
		$\forall x, y \in \mathbb{N}.\ \forall l, k \in \mathbb{L}.(\mathsf{pref}(\mathsf{cons}(x, l), \mathsf{cons}(y, k))$
		$\leftrightarrow (x \simeq y \land \mathsf{pref}(l, k)))$
		$\forall l \in \mathbb{L}.\ \mathsf{suff}(\mathsf{nil}, l)$
		$\forall x \in \mathbb{N}.\ \forall l \in \mathbb{L}.\ \neg(\mathsf{suff}(\mathsf{cons}(x, l), \mathsf{nil}))$
		$\forall x \in \mathbb{N}.\ \forall l, k \in \mathbb{L}.\ (\mathsf{suff}(k, l) \to \mathsf{suff}(k, \mathsf{cons}(x, l)))$
		$\forall x \in \mathbb{N}.\ \forall l, k \in \mathbb{L}.\ (\mathsf{suff}(\mathsf{cons}(x, k), l) \to \mathsf{suff}(k, l))$
		$\forall x \in \mathbb{N}.\ \neg \mathsf{in}_{\mathbb{L}}(x, \mathsf{nil})$
		$\forall x, y \in \mathbb{N}.\ \forall l \in \mathbb{L}.\ (\mathsf{in}_{\mathbb{L}}(x, \mathsf{cons}(y, l)) \leftrightarrow (\mathsf{in}_{\mathbb{L}}(x, l) \lor x \simeq y))$
Binary trees $\mathbb{BT}$	Constructors:	$\mathsf{Nil} : \mathbb{BT}, \qquad \mathsf{node} : \mathbb{BT} \times \mathbb{N} \times \mathbb{BT} \to \mathbb{BT}$
	Symbols:	$\mathsf{in}_{\mathbb{BT}} : \mathbb{N} \times \mathbb{L} \to \mathrm{Bool}, \qquad \mathsf{flat} : \mathbb{BT} \to \mathbb{L}$
	Axioms:	$\forall x \in \mathbb{N}.\ \neg \mathsf{in}_{\mathbb{BT}}(x, \mathsf{Nil})$
		$\forall x, y \in \mathbb{N}.\ \forall l, r \in \mathbb{BT}.\ (\mathsf{in}_{\mathbb{BT}}(x, \mathsf{node}(l, y, r))$
		$\leftrightarrow (\mathsf{in}_{\mathbb{BT}}(x, l) \lor \mathsf{in}_{\mathbb{BT}}(x, r) \lor x \simeq y))$
		$\mathsf{flat}(\mathsf{Nil}) \simeq \mathsf{nil}$
		$\forall x \in \mathbb{N}.\ \forall l, r \in \mathbb{BT}.\ \mathsf{flat}(\mathsf{node}(l, x, r)) \simeq \mathsf{flat}(l) \mathbin{+\!\!+} \mathsf{cons}(x, \mathsf{flat}(r))$

Figure 2.1: Term algebras of $\mathbb{N}$, $\mathbb{L}$, and $\mathbb{BT}$, together with additional symbols and axioms.

For reference we include the definitions of these term algebras, extended by additional function and predicate symbols, in Figure 2.1 We denote the sorts of symbols and terms by : (colon), e.g., $f : \tau \to \alpha$ is a function symbol with domain τ and range α. To emphasize the sort τ of a quantified variable x, we write $\forall x \in \tau$ or $\exists x \in \tau$.

In particular, we will deal with the functions and predicates $+_{\mathbb{N}}, \cdot_{\mathbb{N}}, \mathsf{half}, \leq_{\mathbb{N}}$ for $\mathbb{N}$ denoting addition, multiplication, floored division by two, and less-or-equal relation; $+\!\!+, \mathsf{len}, \mathsf{pref}, \mathsf{suff}, \mathsf{in}_{\mathbb{L}}$ for $\mathbb{L}$, denoting the list concatenation, length of a list, prefix and suffix relations, and the member relation; and $\mathsf{in}_{\mathbb{BT}}, \mathsf{flat}$, denoting the member relation and flattening of a tree to a list, respectively. These additional symbols are axiomatized by first-order formulas corresponding to their recursive definitions, shown in Figure 2.1. Further, we consider $t <_{\mathbb{N}} s$ to be a syntactic shorthand for $\neg(s \leq_{\mathbb{N}} t)$.

For a term algebra sort τ, we denote its constructors with Σ_τ. We fix an arbitrary ordering on the constructors, and denote the i-th constructor in the order by c_i, i.e., $\Sigma_\tau = \{c_1, \dots, c_{|\Sigma_\tau|}\}$. For each c_i, we denote its arity with n_{c_i}. We denote with P_{c_i} the set of argument positions of c_i of the sort τ. We define terminating recursive functions $f : \tau_1 \times \dots \times \tau_n \to \alpha$, where τ_i is a term algebra type, and each τ_k for $k \neq i$ and α are arbitrary types, by providing a set of equalities

$$\{f(y_1, \dots, y_{i-1}, c(\overline{x}), y_{i+1}, \dots, y_n) \simeq \\ t_c[\overline{x}, f(y_1, \dots, y_{i-1}, x_{j_1}, y_{i+1}, \dots, y_n), \dots, f(y_1, \dots, y_{i-1}, x_{j_{|P_c|}}, y_{i+1}, \dots, y_n)]\}_{c \in \Sigma_\tau},$$

where $P_c = \{j_1, \dots, j_{|P_c|}\}$, and each t_c is a term of type α containing no occurrences of f except for the distinguished ones. When we define more than one terminating recursive function, we do it such that there is no circular dependency: the functions can be ordered as $f_1, \dots, f_n$ such that the equalities defining each f_i contain no occurrences of f_j for any $j > i$. An example of such terminating recursive functions are $+_{\mathbb{N}}, \cdot_{\mathbb{N}}$ defined by the first four axioms of $\mathbb{N}$ in Figure 2.1.

Further, we also assume a distinguished *integer sort*, denoted by $\mathbb{Z}$. When we use standard integer predicates $<_{\mathbb{Z}}$, $\leq_{\mathbb{Z}}$, $>_{\mathbb{Z}}$, $\geq_{\mathbb{Z}}$, functions $+_{\mathbb{Z}}, -_{\mathbb{Z}}, \dots$ and constants $0, 1, 2, \dots$, we assume that they denote the corresponding interpreted integer predicates and functions with their standard interpretations.

Note that some function and relation symbols for $\mathbb{N}, \mathbb{L}, \mathbb{BT}$ and $\mathbb{Z}$ only differ in the subscript. When the sort is clear from the context, we drop the subscript.

We use the standard semantics for FOL. Constants $\top, \bot$ are interpreted as themselves. We only consider the standard models of term algebras, and interpret all ground terms consisting only of term algebra constructors as themselves. For an interpretation function I, we denote the interpretation of a variable x, function symbol f, and a predicate symbol p by x^I, f^I, p^I, respectively. We use the notation E^I, F^I also for the interpretation of expressions E and formulas F, respectively. Further, for a variable or a constant a and a value o, we denote by $I\{a \mapsto o\}$ the interpretation function I' such that $a^{I'} = o$ and $b^{I'} = b^I$ for any constant or variable $b \neq a$. We write $F_1, \dots, F_n \vdash G_1, \dots, G_m$ to denote

Algorithm 2.1: The Saturation Loop.

```
1 initial set of clauses $S := \{\mathsf{cnf}(\neg F)\}$
2 repeat
3    Select clause $G \in S$
4    Derive consequences $C_1, \dots, C_n$ of $G$ and formulas from $S$ using rules of $\mathcal{I}$
5    $S := S \cup \{C_1, \dots, C_n\}$
6    if $\square \in S$ then return $F$ is valid
8 return $F$ is not valid
```

that $F_1 \land \ldots \land F_n \rightarrow G_1 \lor \ldots \lor G_m$ is valid, and extend the notation also to validity modulo a theory T.

We recall the standard notion of λ-expressions. Let t be a term and x a variable. Then $\lambda x.t$ denotes a *λ-expression*. For any interpretation I, we define $(\lambda x.t)^I$ as the function f given by $f(o) = t^{I\{x \mapsto o\}}$ for any value o. Moreover, we extend the notation of λ-expressions to also bind constants. Let c be a constant, then $\lambda c.t$ also denotes a *λ-expression*, and its interpretation $(\lambda c.t)^I$ is the function f given by $f(o) = t^{I\{c \mapsto o\}}$ for any value o.

Given any b, we write $a := b$ to denote assignment of b into a. In particular, given an expression E, we write $a := E$ to emphasize that we are assigning E into a, and thus for a given expression $E'[a]$ we obtain $E'[E]$.

2.1 Saturation

Saturation-based proof search implements *proving by refutation* [KV13]: to prove validity of F, a saturation algorithm establishes unsatisfiability of $\neg F$. First-order theorem provers work with clauses, rather than with arbitrary formulas. To prove a formula F, first-order provers negate F which is further skolemized and converted to CNF. The provers thus obtain $\mathsf{cnf}(\neg F)$, which forms a set S of initial clauses. First-order provers then *saturate* S by computing logical consequences of S with respect to a sound inference system $\mathcal{I}$. The saturated set of S is called the *closure* of S and the process of computing the closure of S is called *saturation*. We refer to the set S throughout saturation as the *search space*. If the closure of S contains the empty clause $\square$, the original set S of clauses is unsatisfiable, and hence the formula F is valid.

We show a simplified saturation algorithm for a sound inference system $\mathcal{I}$ in Algorithm 2.1, with a goal F as input.

We may extend the set S of initial clauses with additional clauses $C_1, \dots, C_n$. If C is derived by saturating this extended set, we say C is derived from S *under additional assumptions* $C_1, \dots, C_n$.

A way in which first-order theorem provers can handle reasoning with theories of term algebras or integers is by extending the search space with axioms and introducing additional inference rules. The axioms for term algebras include domain closure, injectivity, distinctness and acyclicity – a detailed definition of these axioms can be found in [RV01, KRV17]. The work [KRV17] addresses the challenge of automating proving term algebras properties given the fact that the acyclicity axiom is not finitely axiomatizable. For reasoning with the theory of integer arithmetic, see [KV13, RSV18, RSV21, KKR^{+}23].

One of the keys to the efficiency of saturation-based theorem proving is *clause splitting*, with the leading approach being the Avatar architecture [Vor14, BRSV16]. The main idea of splitting is as follows. Let S be a set of clauses and $C_1 \vee C_2$ a clause such that C_1, C_2 have disjoint sets of variables. We call such clauses C_1, C_2 the *components* of $C_1 \vee C_2$. Then $S \cup \{C_1 \vee C_2\}$ is unsatisfiable iff both $S \cup \{C_1\}$ and $S \cup \{C_2\}$ are unsatisfiable. Therefore, instead of checking satisfiability of a set of large clauses, we check the satisfiability of multiple sets of smaller clauses. Avatar implements this idea by using an interplay between a saturation-based first-order theorem prover and a SAT/SMT solver. The SAT/SMT solver finds a set of clause components, satisfiability of which implies satisfiability of all split clauses. These components, called *assertions*, are then used by the theorem prover for further derivations in saturation. All clauses derived using assertions $C_1, \ldots, C_n$ are called *clauses with assertion* $C_1, \ldots, C_n$. Finally, note that when Avatar uses an SMT solver, the solver can also check the theory-consistency of the assertions, and thus facilitate reasoning with theories.

2.2 Superposition

The *superposition calculus*, denoted as $\mathbb{S}\text{up}$, is the most common inference system used by saturation-based provers for first-order logic with equality [NR01]. An example of such a prover is the theorem prover Vampire [KV13], in which we implement the contributions described in this thesis. An overview of the inference rules of $\mathbb{S}\text{up}$ is given in Figure 2.2. In the derivations we show in this thesis, we indicate how was each formula derived by listing the (abbreviated) name of the applied rule in brackets.

The $\mathbb{S}\text{up}$ calculus is parametrized by a *simplification ordering* $\succ$ on terms and a *selection function*, which selects in each non-empty clause a non-empty subset of literals (possibly also positive literals). We denote selected literals by underlining them. An inference rule can be applied on the given premise(s) if the literals that are underlined in the rule are also selected in the premise(s). The superposition calculus $\mathbb{S}\text{up}$ is *sound* (if $\square$ is derived from F, then F is unsatisfiable) and, for a certain class of selection functions, it is also *refutationally complete* (if F is unsatisfiable, then $\square$ can be derived from it).

2.3 Induction in Saturation

Inductive reasoning has been integrated into saturation in [RV19], and further extended in [HHK^{+}20, HKV21, HHKV21, HKRV22], where the former two papers form the first

Superposition (Sup):

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

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

Factoring (Fac):

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

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

Binary resolution (BR):

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

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

Equality resolution (ER):

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

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

Equality factoring (EF):

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

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

Figure 2.2: The superposition calculus $\mathbb{S}\mathrm{up}$. Underlined literals are selected. In proofs, we denote the rules we use by the abbreviations in parentheses.

part of this thesis. For a survey, see also [HHK+22].

The main idea in this body of work is to apply induction by *theory lemma generation*: based on already derived formulas, generate a suitable induction axiom and add it to the search space. To this end, the following induction rule is used:

$$\frac{\overline{L}[t] \lor C}{F \to \forall x.L[x]}\ (\mathsf{Ind}), \tag{2.1}$$

where $L[t]$ is a ground literal, C is a clause, and $F \to \forall x.L[x]$ is a valid induction axiom. The conclusion of the Ind rule is skolemized to obtain *induction formula* and then clausified, yielding clauses $\mathsf{cnf}(\neg F) \lor L[x]$. These clauses are resolved with the premise $\overline{L}[t] \lor C$ immediately after applying the Ind rule and the resulting clauses $\mathsf{cnf}(\neg F) \lor C$ are added to the search space. Note that in some of the works, the clausification and resolution steps are explicitly captured by formulating the induction rule as

$$\frac{\overline{L}[t] \lor C}{\mathsf{cnf}(\neg F) \lor L[x]} \quad \text{or} \quad \frac{\overline{L}[t] \lor C}{\mathsf{cnf}(\neg F) \lor C}\ .$$

In this thesis, we will stick to the formulation of Ind from (2.1), but we emphasize that the conclusion of the rule is always clausified and resolved with the premise of the rule.

An *induction schema* is a collection of induction axioms. Each induction schema we consider is the set of first-order instances of some valid higher-order formula. An example of a valid induction schema is the *structural induction schema for natural numbers* [RV19], where $G[x]$ is any closed formula with x a variable of the natural number sort:

$$(G[\mathsf{0}] \land \forall y.(G[y] \rightarrow G[\mathsf{s}(y)])) \rightarrow \forall x.G[x] \tag{2.2}$$

Informally, the schema expresses that if the base case holds, and if the induction step holds, then $G[x]$ holds for all possible values of x. When we instantiate the schema with $G[x] := L[x]$, we obtain an axiom that can be used in Ind. Note that we can also use a complex formula $G[t]$ in place of the literal $L[t]$ in Ind, obtaining a more involved rule, possibly with multiple premises, similarly to a *mutli-clause induction rule* [HHKV21] or a *induction with arbitrary formulas* [HKRV22].

The CNF of the structural induction axiom instantiated by $L[x]$ is

$$\neg L[\mathsf{0}] \lor L[\sigma] \lor L[x]$$
$$\neg L[\mathsf{0}] \lor \neg L[\mathsf{s}(\sigma)] \lor L[x],$$

where σ is the skolem constant corresponding to y from the axiom. After binary resolution with the premise of the induction rule $\overline{L}[t] \lor C$ we obtain:

$$\neg L[\mathsf{0}] \lor L[\sigma] \lor C$$
$$\neg L[\mathsf{0}] \lor \neg L[\mathsf{s}(\sigma)] \lor C$$

These are the clauses that are added to the search space.

Compared with inductive provers such as [BSvH$^+$93, CJRS12, SDE12, BM79, PCI$^+$20], the approach used in [RV19] and this thesis automates induction by integrating it directly in superposition-based proof search, without relying on rewrite rules and external heuristics for generating auxiliary inductive lemmas/subgoals. This approach is also conceptually different from the previous attempts to use induction with superposition [KP13, Cru17, EP20], as we are not restricted to specific clause splitting algorithms and heuristics used in [Cru17], nor are we limited to induction over term algebras with the subterm ordering in [EP20]. As a result, we stay within the standard saturation framework and do not have to introduce constraint clauses, additional predicates or change the notion of redundancy as in [EP20].

In this thesis, we will sometimes write “this problem requires induction”. This should not be regarded as a formal statement: this property is not easy to formalize in general and it is possible that some of these problems can be proved by certain combinations of decision procedures, first-order theorem proving with uninterpreted functions, and axiomatization of interpreted functions. However, when we make such statements, one can see that these problems have relatively simple proofs involving induction and cannot be proved by existing provers without induction.

2.4 Answer Literals

Answer literals [Gre69] provide a question answering technique for tracking substitutions into given variables throughout the proof. Suppose we want to find a witness for the validity of the formula

$$\exists y.F[y]. \tag{2.3}$$

Within saturation-based proving, we first derive the skolemized negation of (2.3) and add an *answer literal* using a fresh predicate ans with argument y, yielding

$$\forall y.(\neg F[y] \vee \mathsf{ans}(y)). \tag{2.4}$$

We then saturate the CNF of (2.4), while ensuring that answer literals are not selected for performing inferences. If the clause $\mathsf{ans}(t_1) \vee \ldots \vee \mathsf{ans}(t_m)$ is derived during saturation, note that this clause contains only answer literals in addition to the empty clause; hence, in this case we proved unsatisfiability of $\forall y.\neg F[y]$, implying validity of (2.3). Moreover, $t_1, \ldots, t_m$ provides a *disjuntive answer*, i.e. witness, for the validity of (2.3); that is, $F[t_1] \vee \ldots \vee F[t_m]$ holds [Kun96]. In particular, if we derive the clause $\mathsf{ans}(t)$ during saturation, we found a *definite answer* t for (2.3), namely $F[t]$ is valid.

Answer literals with $\mathtt{if-then-else}$. The derivation of disjunctive answers can be avoided by modifying the inference rules to only derive clauses containing at most one answer literal. One such modification is given within the A(R)-calculus for binary resolution [Tam95]. The calculus is parametrized by a strongly liftable term restriction R, i.e., a restriction such that if $R(t\theta)$ holds for any term t and substitution θ, then also $R(t)$ holds. The A(R)-calculus replaces the binary resolution rule when both premises contain an answer literal by the following A-resolution rule:

$$\frac{A \vee C \vee \mathsf{ans}(r) \quad \neg A' \vee C' \vee \mathsf{ans}(r')}{(C \vee C' \vee \mathsf{ans}(\mathtt{if}\ A\ \mathtt{then}\ r'\ \mathtt{else}\ r))\theta}\ (A\text{-}\mathsf{resolution}),$$

where $\theta := \mathsf{mgu}(A, A')$ and the restriction $R(\mathtt{if}\ A\ \mathtt{then}\ r'\ \mathtt{else}\ r)$ holds. We illustrate the use of the A(R)-calculus by the following example from [Reg18].

Example 2.1. Let $\mathsf{arcade}, \mathsf{vampire}$ be constants, $\mathsf{sunday}, \mathsf{monday}$ boolean constants, and $\mathsf{workshop}$ a unary predicate. The example models a conference, where the arcade workshop takes place on sunday and $\mathsf{vampire}$ workshop on monday, and states that it is either sunday or monday. The specification then asks for a workshop:

$$\begin{aligned}
\text{axioms: } & \mathsf{sunday} \rightarrow \mathsf{workshop}(\mathsf{arcade}) \\
& \mathsf{monday} \rightarrow \mathsf{workshop}(\mathsf{vampire}) \\
& \mathsf{sunday} \vee \mathsf{monday} \\
\text{specification: } & \exists x.\mathsf{workshop}(x)
\end{aligned}$$

A possible definite answer for this input would be:

$$\mathtt{if}\ \mathsf{workshop}(\mathsf{arcade})\ \mathtt{then}\ \mathsf{arcade}\ \mathtt{else}\ \mathsf{vampire}$$

However, it is disputable if this answer is helpful: after all, if we could evaluate whether a condition $\mathsf{workshop}(\cdot)$ holds ourselves, we would not need to pose the query $\exists x.\mathsf{workshop}(x)$ at all. Therefore, we define the restriction $R(t)$ to be true iff t does not contain the symbol $\mathsf{workshop}$. With this restriction, we derive a definite answer using the A(R)-calculus. For each clause in the derivation, we list how the clause has been derived. For example, clause 5 is the result of binary resolution (BR) applied on clauses 1 and 3.

1. $\neg\mathsf{workshop}(x) \vee \mathsf{ans}(x)$ [preprocessed specification with answer literal]
2. $\mathsf{sunday} \vee \mathsf{monday}$ [input axiom]
3. $\neg\mathsf{sunday} \vee \mathsf{workshop}(\mathsf{arcade})$ [input axiom]
4. $\neg\mathsf{monday} \vee \mathsf{workshop}(\mathsf{vampire})$ [input axiom]
5. $\neg\mathsf{sunday} \vee \mathsf{ans}(\mathsf{arcade})$ [BR 1, 3]
6. $\neg\mathsf{monday} \vee \mathsf{ans}(\mathsf{vampire})$ [BR 1, 4]
7. $\mathsf{sunday} \vee \mathsf{ans}(\mathsf{vampire})$ [BR 2, 6]
8. $\mathsf{ans}(\texttt{if}\ \mathsf{sunday}\ \texttt{then}\ \mathsf{arcade}\ \texttt{else}\ \mathsf{vampire})$ [A-resolution 5, 7]

Thus, the definite answer is $\texttt{if}\ \mathsf{sunday}\ \texttt{then}\ \mathsf{arcade}\ \texttt{else}\ \mathsf{vampire}$. □

CHAPTER 3

Induction with Generalization

The contributions of this chapter are based on:
Márton Hajdu, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Induction with Generalization in Superposition Reasoning. In Christoph Benzmüller and Bruce Miller, editors, Proc. of CICM, *volume 12236 of* LNCS, *pages 123–137, Cham, 2020. Springer [HHK+20]*

In this chapter we extend inductive reasoning in saturation-based framework by introducing a new rule for induction with generalization. The rule adds induction axioms for proving generalizations of the literals appearing during proof search.

Given a formula (goal) F, it is common in inductive theorem proving to try to prove a more general goal instead [BM79]. This makes no sense in saturation-based theorem proving, which is not based on a goal-subgoal architecture. As we aim to automate and generalize inductive reasoning within saturation-based proof search, *our work follows a different approach than the one used in inductive theorem provers.* Namely, our methodology (Section 3.2) picks up a formula F (not necessarily the goal) in the search space and *adds to the search space new induction axioms with generalization*, that is, instances of generalized induction schemas, aiming at proving both $\neg F$ and a more general formula than $\neg F$.

We open this chapter by giving a concrete example motivating our approach in Section 3.1. While we use $\mathbb{N}$ for illustration, we note that our approach can be used for proving properties over any other theories with various forms of induction. The example illustrates the advantage of induction with generalization in saturation-based proof search. We then present a new inference rule for first-order superposition reasoning, called *induction with generalization* (Section 3.2). Our work extends [RV19] by proving properties with multiple occurrences of the same induction term and by instantiating induction axioms with logically stronger versions of the property being proved.

Further, in the following chapters we describe our implementation and experimental evaluation of the method from this chapter (see Section 6.1), and we present a new dataset of benchmarks focused on induction with generalization (see Section 5.2.1). Our experiments show that our new approach solves many problems that other existing systems cannot solve.

We note that in this chapter we only work with the sort of $\mathbb{N}$ and thus drop the data type subscript from the function symbol $+_{\mathbb{N}}$.

3.1 Motivating Example

We motivate our approach to induction with generalization by variations of the associativity property of addition over $\mathbb{N}$.

Example 3.1. Consider the following formula expressing the associativity of addition:

$$\forall x, y, z \in \mathbb{N}.\ x + (y + z) \simeq (x + y) + z \tag{3.1}$$

We preprocess (3.1) by negating and skolemizing it, obtaining

$$\sigma_1 + (\sigma_2 + \sigma_3) \not\simeq (\sigma_1 + \sigma_2) + \sigma_3, \tag{3.2}$$

where $\sigma_1, \sigma_2, \sigma_3$ are fresh skolem constants used to skolemize x, y, z, respectively. The induction approach introduced in [RV19] uses (3.2) to instantiate structural induction schema (2.2) resulting in the following axiom:

$$\begin{aligned}&(\mathsf{0} + (\sigma_2 + \sigma_3) \simeq (\mathsf{0} + \sigma_2) + \sigma_3 \land \\ &\qquad \forall y.(y + (\sigma_2 + \sigma_3) \simeq (y + \sigma_2) + \sigma_3 \rightarrow \mathsf{s}(y) + (\sigma_2 + \sigma_3) \simeq (\mathsf{s}(y) + \sigma_2) + \sigma_3)) \\ &\quad \rightarrow \forall x.(x + (\sigma_2 + \sigma_3) \simeq (x + \sigma_2) + \sigma_3)\end{aligned} \tag{3.3}$$

Using this induction axiom we obtain a refutational proof of (3.1), with the main steps discussed below:

1. $\sigma_1 + (\sigma_2 + \sigma_3) \not\simeq (\sigma_1 + \sigma_2) + \sigma_3$ [preprocessed input]
2. $\mathsf{0} + (\sigma_2 + \sigma_3) \not\simeq (\mathsf{0} + \sigma_2) + \sigma_3 \lor \sigma + (\sigma_2 + \sigma_3) \simeq (\sigma + \sigma_2) + \sigma_3 \lor$ $x + (\sigma_2 + \sigma_3) \simeq (x + \sigma_2) + \sigma_3$ [Ind with (3.3)]
3. $\mathsf{0} + (\sigma_2 + \sigma_3) \not\simeq (\mathsf{0} + \sigma_2) + \sigma_3 \lor \mathsf{s}(\sigma) + (\sigma_2 + \sigma_3) \not\simeq (\mathsf{s}(\sigma) + \sigma_2) + \sigma_3 \lor$ $x + (\sigma_2 + \sigma_3) \simeq (x + \sigma_2) + \sigma_3$ [Ind with (3.3)]
4. $\mathsf{0} + (\sigma_2 + \sigma_3) \not\simeq (\mathsf{0} + \sigma_2) + \sigma_3 \lor \sigma + (\sigma_2 + \sigma_3) \simeq (\sigma + \sigma_2) + \sigma_3$ [BR 1, 2]
5. $\mathsf{0} + (\sigma_2 + \sigma_3) \not\simeq (\mathsf{0} + \sigma_2) + \sigma_3 \lor \mathsf{s}(\sigma) + (\sigma_2 + \sigma_3) \not\simeq (\mathsf{s}(\sigma) + \sigma_2) + \sigma_3$ [BR 1, 3]
6. $\mathsf{0} + (\sigma_2 + \sigma_3) \not\simeq (\mathsf{0} + \sigma_2) + \sigma_3 \lor \mathsf{s}(\sigma + (\sigma_2 + \sigma_3)) \not\simeq \mathsf{s}((\sigma + \sigma_2) + \sigma_3)$ [5, axiom of +]

7. $\mathsf{0} + (\sigma_2 + \sigma_3) \not\simeq (\mathsf{0} + \sigma_2) + \sigma_3 \vee \sigma + (\sigma_2 + \sigma_3) \not\simeq (\sigma + \sigma_2) + \sigma_3$ [injectivity 6]
8. $\mathsf{0} + (\sigma_2 + \sigma_3) \not\simeq (\mathsf{0} + \sigma_2) + \sigma_3$ [BR 4, 7]
9. $\sigma_2 + \sigma_3 \not\simeq \sigma_2 + \sigma_3$ [8, axiom of +]
10. $\square$ [trivial inequality removal 9]

Clauses 2 and 3 are the CNF of induction axiom (3.3). These clauses are resolved against clause 1, yielding clauses 4 and 5. Clause 6 originates by repeated demodulation into 5 using the second axiom of + from Figure 2.1 over $\mathbb{N}$. Further, 7 is derived from 6 by using the injectivity property of term algebras and 8 is a resolvent of 4 and 7. Clause 9 is then derived by repeated demodulation into 8, using the the first axiom of + from Figure 2.1 over $\mathbb{N}$. By removing the trivial inequality from 9, we finally derive the empty clause 10. $\square$

While (3.1) was easily proved using Ind, the next example shows the limitations of this rule.

Example 3.2. Consider now the following instance of associativity property (3.1):

$$\forall x \in \mathbb{N}.\ x + (x + x) \simeq (x + x) + x \tag{3.4}$$

While (3.4) is an instance of (3.1), we cannot prove it using the same approach. Let us explain why this is the case. By instantiating induction schema (2.2) using (3.4), we get:

$$\begin{aligned}&(\mathsf{0} + (\mathsf{0} + \mathsf{0}) \simeq (\mathsf{0} + \mathsf{0}) + \mathsf{0} \wedge \\ &\qquad \forall y.(y + (y + y) \simeq (y + y) + y \rightarrow \mathsf{s}(y) + (\mathsf{s}(y) + \mathsf{s}(y)) \simeq (\mathsf{s}(y) + \mathsf{s}(y)) + \mathsf{s}(y))) \\ &\ \rightarrow \forall x.(x + (x + x) \simeq (x + x) + x)\end{aligned} \tag{3.5}$$

After resolving this axiom with the skolemized negation of (3.4), we get the following two clauses:

$$\mathsf{0} + (\mathsf{0} + \mathsf{0}) \not\simeq (\mathsf{0} + \mathsf{0}) + \mathsf{0}\ \vee\ \sigma + (\sigma + \sigma) \simeq (\sigma + \sigma) + \sigma \tag{3.6}$$

$$\mathsf{0} + (\mathsf{0} + \mathsf{0}) \not\simeq (\mathsf{0} + \mathsf{0}) + \mathsf{0}\ \vee\ \mathsf{s}(\sigma) + (\mathsf{s}(\sigma) + \mathsf{s}(\sigma)) \not\simeq (\mathsf{s}(\sigma) + \mathsf{s}(\sigma)) + \mathsf{s}(\sigma) \tag{3.7}$$

While the first literals of (3.6) and (3.7) are easily resolved using axioms of +, not much can be done with the latter literals. We can only apply repeated demodulations over the second literal of (3.7) using axioms of + and the injectivity property of term algebras, yielding $\sigma + s(\sigma + s(\sigma)) \not\simeq (\sigma + s(\sigma)) + s(\sigma)$. No further inference over this formula can be applied, in particular, it cannot be resolved against the second literal of (3.6). Hence, the approach of [RV19] fails proving (3.4). $\square$

The existing approaches to induction also suffer from the same problem. For example [BM79, PCI$^+$20, BCD$^+$11, SDE12, Cru17], can prove property (3.1) but fail to prove its weaker instance (3.4). The common recipe in inductive theorem proving [BM79] is to try to prove (3.1) in addition to trying to prove (3.4).

Interestingly, in saturation-based theorem proving we can do better. If we follow the common recipe, we would add a generalized goal and then an induction axiom for it. Instead, we only add the induction axiom instance corresponding to the generalized goal without adding the extra goal, which results in a smaller number of clauses. More precisely, in addition to the instance of the induction schema corresponding to (3.4), we also add an instance corresponding to $\forall x, y \in \mathbb{N}.\ x + (y + y) \simeq (x + y) + y$. We call this new inference rule *induction with generalization*.

3.2 Induction with Generalization

Following [RV19], we consider an *induction axiom* to be any formula of the form $premise \rightarrow \forall x.L[x]$, valid in the underlying theory, such as the theory of term algebras. An example of an induction axiom is structural induction axiom (3.3). We recall rule Ind [RV19], where a ground literal $\overline{L}[t]$ appearing in the proof search triggers the addition of the corresponding induction axiom $premise \rightarrow \forall x.L[x]$ to the search space:

$$\frac{\overline{L}[t] \vee C}{premise \rightarrow \forall x.L[x]}\ (\mathsf{Ind}),$$

where $L[x]$ is obtained from $L[t]$ by replacing *all* occurrences of t by x. Examples of axioms added by instances of Ind are (3.3) and (3.5).

While addition of a large number of such formulas may seem to blow up the search space, in practice Vampire handles such addition with little overhead, resulting in finding proofs containing nearly 150 induction inferences [RV19]. The reason why the overhead of adding structural induction axioms is small is explained in [RV20]: the added clauses only contain one variable (the x in $L[x]$), and the clauses containing this literal are immediately subsumed by a ground clause. The net result is adding a small number of ground clauses, which are especially easy to handle in the Avatar architecture implemented in Vampire.

Induction with generalization. In a nutshell, given a goal, we add an induction axiom corresponding to a more general one. The rule can be formulated in the same way as Ind, yet with a different conclusion:

$$\frac{\neg L[t] \vee C}{\mathsf{cnf}(premise' \rightarrow \forall x.L'[x])}\ (\mathsf{IndGen}), \tag{3.8}$$

where $L'[x]$ is obtained from $L[t]$ by replacing *some* occurrences of t by x, and $premise'$ is the premise corresponding to $L'[x]$. Both induction rules are obviously sound because their conclusions are constructed such that they are valid in the underlying theory.

To implement IndGen, if a clause selected for inferences contains a ground literal $\neg L[t]$ having more than one occurrence of t, we should select a non-empty subset of occurrences of t in $L[t]$, select an induction axiom corresponding to this subset, and then apply the rule.

Example 3.3. We again consider proving (3.4) from Example 3.2. Suppose that t is σ_1 and $\neg L[t]$ is $\sigma_1 + (\sigma_1 + \sigma_1) \not\simeq (\sigma_1 + \sigma_1) + \sigma_1$, which is obtained by negating and skolemizing (3.4). Then by applying IndGen we can add the induction axiom

$$\begin{aligned}&(0 + (\sigma_1 + \sigma_1) \simeq (0 + \sigma_1) + \sigma_1 \wedge \\ &\quad \forall y.(y + (\sigma_1 + \sigma_1) \simeq (y + \sigma_1) + \sigma_1 \rightarrow \mathsf{s}(y) + (\sigma_1 + \sigma_1) \simeq (\mathsf{s}(y) + \sigma_1) + \sigma_1)) \\ &\rightarrow \forall x.(x + (\sigma_1 + \sigma_1) \simeq (x + \sigma_1) + \sigma_1),\end{aligned} \tag{3.9}$$

which is different from (3.5). When we add this formula, we can derive the empty clause from (3.4) in the same way as in the proof of (3.1) from Section 3.1. □

Saturation with induction with generalization. The main questions to answer when applying induction with generalization is which occurrences of the induction term in the induction literal we should choose.

Generally, if the subterm t occurs n times in the premise, there are $2^n - 1$ ways of applying the rule, all potentially resulting in *formulas not implying each other*. Thus, an obvious heuristic to use *all* non-empty subsets may result in too many formulas. For example, $\sigma_1 + (\sigma_1 + \sigma_1) \not\simeq (\sigma_1 + \sigma_1) + \sigma_1$ would result in adding 63 induction formulas.

Another simple heuristic is to restrict the number of occurrences selected as induction term to a fixed number. This strategy reduces the number of applications of induction at the cost of losing proofs that would need subsets of cardinality larger than the limit. Finding possible heuristics for selecting specific subsets for common cases of literals can be the subject of future work, especially interesting in proof assistants in mathematics, and we note this challenge was later addressed in [HHKV21].

Note that some of the conclusions of (3.8) can, in turn, have many children obtained by induction with generalization. Our experiments in Section 6.1 show that, even when we generate all possible children, Vampire can still solve large examples with more than 10 occurrences of the same induction variable, again thanks to the effect that, for each application of induction, only a small number of ground clauses turn out to be added to the search space.

We therefore believe that our work can potentially be also useful for larger examples, and even in cases when the inductive property to be proved is embedded in a larger context.

CHAPTER 4

Integer Induction

The contributions of this chapter are based on:
Petra Hozzová, Laura Kovács, and Andrei Voronkov. Integer Induction in Saturation. In André Platzer and Geoff Sutcliffe, editors, Proc. of CADE, *volume 12699 of* LNCS, *pages 361–377, Cham, 2021. Springer [HKV21]*

In this chapter we analyze the challenge of automating inductive reasoning with integers and introduce inference rules for integer induction within the saturation framework, including techniques for discovering a suitable base case.

We note that in this chapter we only work with the sort of $\mathbb{Z}$ and thus drop the data type subscript from the predicate and function symbols $\leq_{\mathbb{Z}}, <_{\mathbb{Z}}, \geq_{\mathbb{Z}}, >_{\mathbb{Z}}, +_{\mathbb{Z}}, \cdot_{\mathbb{Z}}, -_{\mathbb{Z}}$.

One of the most commonly used data types in imperative/functional programs are integers. For example, iterating over arrays in imperative programs or recursively computing sums in functional programs include integer-valued program variables, as illustrated in Figure 4.1. While for many uses of integers in programming we only need to consider non-negative integers, there are also applications where integers are essential, for example, reasoning about memory. To formally prove functional correctness of such and similar programs, reasoning about integers is indispensable but so is handling some sort of induction over integers. In this chapter we address these two reasoning challenges and fully automate inductive reasoning with integers within saturation-based theorem proving.

The works of [Cru17, RV19] and Chapter 3 focused on induction on inductively defined data types, also called algebraic data types [KRV17], such as natural numbers or lists. However, automating *integer induction*, that is, induction on integers, has not yet been addressed sufficiently.

While natural numbers have a well-founded order and induction over this order is very useful in automated inductive theorem proving, the standard order on integers is not well-founded, so it cannot be directly used as the induction ordering. In this chapter we will use the observation that the standard ordering $<$ is well-founded on every set of integers having a lower bound b and likewise, the inverse $>$ of this ordering is well-founded on every set of integers having an upper bound b. This gives us two induction rules on such integer subsets: induction (with the base case b) using $<$ and induction (with the base case b) using $>$, respectively, to prove that a property holds for all integers $\geq b$ and $\leq b$, respectively. We define these induction rules as *upward* and *downward induction rules with symbolic bounds*, respectively. We also consider two variations of these rules over finite integer intervals and refer to such rules as *interval upward* and *interval downward induction rules with symbolic bounds*, respectively.

For natural numbers, $\mathsf{0}$ is an obvious base case candidate, which also turns out to be successful in the theorem proving practice. Analogously, 0 is a natural base case candidate for integer induction. Nonetheless, in this chapter we present some natural problems for which neither 0 nor any concrete integer is a good base case.

In Section 4.1 we illustrate our approach by considering properties of the functional and imperative programs of Figure 4.1. Then in Section 4.2 we define four induction rules over integers, called *(interval) downward, respectively upward, induction rules with symbolic bounds*, and prove their soundness. Section 4.3 introduces an extension of superposition calculus by our new integer induction rules. These rules are formulated in the context of saturation-based theorem proving in a way that avoids an immediate combinatorial explosion of the search space. We demonstrate that using this extension, superposition provers can prove integer properties similarly to how humans would do. This extension is especially successful when used together with the Avatar architecture [Vor14], since Avatar helps in reasoning efficiently using constraints coming out of the integer induction rules.

In later chapters we also describe our implementation of this work and its experimental evaluation (see Section 6.2), as well as our associated benchmark set (see Section 5.2.2).

4.1 Motivating Examples

To illustrate problems arising in automating integer induction, let us consider the programs of Figure 4.1. Properties of both programs are specified using assertions expressed in first-order logic, with pre- and post-conditions specified by the keywords **assume** and **assert**, respectively.

Functional programs. The ML-style functional program of Figure 4.1(a) computes the sum $\mathsf{sum}(n, m)$ of integers in the interval $[n, m]$, that is $\sum_{i=n}^{m} i$, where $m \geq n$. The

```
fun sum$(n, m)$ =
   if $n = m$ then $n$
   else $n + \mathsf{sum}(n + 1, m)$;

assert  $\forall n, m \in \mathbb{Z}.(n \leq m \;\to\; 2 \cdot \mathsf{sum}(n, m) \simeq m \cdot (m + 1) - n \cdot (n - 1))$
```

(a) Sum of integers from $[n, m]$.

```
assume  $0 \leq pos < \mathrm{A.size}$

$i := pos$;
while $i + 1 < \mathrm{A.size}$ do
   $\mathrm{A}[i + 1] := \mathrm{A}[i]$;
   $i := i + 1$;
   inv $\forall j \in \mathbb{Z}.(pos \leq j < i \;\to\; \mathsf{val}_\mathrm{A}(j + 1) \simeq \mathsf{val}_\mathrm{A}(j))$
end

assert  $\forall j \in \mathbb{Z}.(pos \leq j < \mathrm{A.size} \;\to\; \mathsf{val}_\mathrm{A}(j) \simeq \mathsf{val}_\mathrm{A}(pos))$
```

(b) Array initialization, with $\mathsf{val}_\mathrm{A}(j)$ denoting $\mathrm{A}[j]$.

Figure 4.1: Motivating examples for inductive reasoning with integers.

function definition uses the following axioms of sum:

$$\forall n \in \mathbb{Z}.(\mathsf{sum}(n, n) \simeq n); \tag{4.1}$$
$$\forall n, m \in \mathbb{Z}.(n \not\simeq m \;\to\; \mathsf{sum}(n, m) \simeq n + \mathsf{sum}(n + 1, m)). \tag{4.2}$$

We should prove the assertion

$$\forall n, m \in \mathbb{Z}.(n \leq m \;\to\; 2 \cdot \mathsf{sum}(n, m) \simeq m \cdot (m + 1) - n \cdot (n - 1)). \tag{4.3}$$

Formally proving (4.3) requires inductive reasoning with both integers and quantifiers. Let $F[x]$ be a formula with one or more occurrences of an integer variable x and b an integer term not containing x. Consider the following formula:

$$F[b] \wedge \forall y \in \mathbb{Z}.(y \leq b \wedge F[y] \to F[y - 1]) \to \forall x \in \mathbb{Z}.(x \leq b \to F[x]) \tag{4.4}$$

This formula is valid. It is similar to the standard structural induction schema on natural numbers (2.2), yet with two essential differences. First, we use $y - 1$ instead of $\mathsf{s}(y)$ and second, we use the term b where for the structural induction on naturals we would use $\mathsf{0}$.

Note that b does not have to be a concrete integer, it can be any term. In the sequel, we will refer to such terms b used in induction rules as *symbolic bounds*.

For proving (4.3) using a theorem prover, we first negate and skolemize (4.3), obtaining the following formula, where σ_n, σ_m are fresh skolem constants:

$$\sigma_n \leq \sigma_m \;\wedge\; 2 \cdot \mathsf{sum}(\sigma_n, \sigma_m) \not\simeq \sigma_m \cdot (\sigma_m + 1) - \sigma_n \cdot (\sigma_n - 1) \tag{4.5}$$

Modern theorem provers implementing linear integer arithmetic and quantifiers can prove unsatisfiability of (4.1), (4.2) and (4.5) in a relatively straightforward way if we also add an instance of induction schema (4.4) with

$$\begin{array}{rcl} F[x] & := & 2 \cdot \mathsf{sum}(x, \sigma_m) \simeq \sigma_m \cdot (\sigma_m + 1) - x \cdot (x - 1); \\ b & := & \sigma_m. \end{array}$$

If we want to automate this kind of reasoning, the main question is finding the corresponding instance of induction schema (4.4), that is, finding the induction target formula $F[x]$ and the (symbolic) bound b.

Imperative programs. The C-style imperative program of Figure 4.1(b) initializes an integer-valued array A starting at the index *pos*. We should prove the assertion stating that all array elements at indices greater than or equal to *pos* are equal to each other. Proving such assertions typically requires loop invariants "summarizing" the loop behavior. One such invariant I is shown in the loop after the keyword **inv**. This invariant I could be derived by existing approaches to invariant generation [FPMG19, GGK20].

The assertion of Figure 4.1(b) is then proved using I, by establishing that the post-condition

$$\forall j \in \mathbb{Z}.(pos \leq j < \mathrm{A.size} \;\rightarrow\; \mathsf{val}_{\mathrm{A}}(j) \simeq \mathsf{val}_{\mathrm{A}}(pos)) \tag{4.6}$$

is a logical consequence of the invariant I and the negation of the loop condition:

$$\forall j \in \mathbb{Z}.(pos \leq j < i \;\rightarrow\; \mathsf{val}_{\mathrm{A}}(j+1) \simeq \mathsf{val}_{\mathrm{A}}(j)) \;\;\wedge\;\; \neg(i + 1 < \mathrm{A.size}) \tag{4.7}$$

Interestingly, modern theorem provers cannot perform such proofs. Similar to the first example, we can use an induction schema for integers formulated as follows:

$$F[b_1] \wedge \forall y \in \mathbb{Z}.(b_1 \leq y < b_2 \wedge F[y] \rightarrow F[y+1]) \rightarrow \forall x \in \mathbb{Z}.(b_1 \leq x \leq b_2 \rightarrow F[x]) \tag{4.8}$$

Here we use two symbolic bounds, b_1 and b_2, to define a finite interval $[b_1, b_2]$ over which we apply induction. If we add an instance of this schema with

$$\begin{array}{rcl} F[x] & := & \mathsf{val}_{\mathrm{A}}(x) \simeq \mathsf{val}_{\mathrm{A}}(pos); \\ b_1 & := & pos; \\ b_2 & := & \mathrm{A.size} - 1, \end{array}$$

then state-of-the-art theorem provers can easily prove that (4.6) is a logical consequence of (4.7) and the corresponding instance of (4.8). For example, CVC4 [BCD+11], Z3 [DMB08] and Vampire prove such an instance in essentially no time. However, similarly to the example of Figure 4.1(a), in order to find such proofs automatically using the induction axiom of (4.8), we need to be able to discover, during the proof search, the induction target formula $F[x]$ and the symbolic bounds b_1, b_2. In what follows, we describe our solution to automating this discovery by integrating integer induction within saturation-based theorem proving.

4.2 Integer Induction Axioms and Rules

In this section we define four induction rules, or induction schemas, on integers. Two of them were already considered in Section 4.1 – namely (4.4) and (4.8).

Definition 4.1 (Downward/Upward Induction)**.** A *downward, respectively upward, induction axiom with symbolic bounds* is any formula of the form

$$F[b] \land \forall y.(y \leq b \land F[y] \to F[y-1]) \to \forall x.(x \leq b \to F[x]); \qquad \textit{(downward)}$$
$$F[b] \land \forall y.(y \geq b \land F[y] \to F[y+1]) \to \forall x.(x \geq b \to F[x]), \qquad \textit{(upward)}$$

respectively, where $F[x]$ is a formula with one or more occurrences of an integer variable x and b is an integer term not containing x. □

Note that (4.4) is a downward induction axiom with symbolic bounds.

Definition 4.2 (Interval Downward/Upward Induction)**.** An *interval downward, respectively upward, induction axiom with symbolic bounds* is any formula of the form

$$F[b_2] \land \forall y.(b_1 < y \leq b_2 \land F[y] \to F[y-1]) \to \forall x.(b_1 \leq x \leq b_2 \to F[x]); \quad \textit{(downward)}$$
$$F[b_1] \land \forall y.(b_1 \leq y < b_2 \land F[y] \to F[y+1]) \to \forall x.(b_1 \leq x \leq b_2 \to F[x]), \quad \textit{(upward)}$$

respectively, where $F[x]$ is a formula with one or more occurrences of an integer variable x and b_1, b_2 are integer terms not containing x. □

Note that (4.8) is an interval upward induction axiom with symbolic bounds.

The main motivation for interval induction rules is their utility in reasoning about loops, as illustrated by the example of Figure 4.1(a). While interval induction can be captured by induction with one bound, it would require additional case analysis, which is not efficient in saturation-based proving practice.

In the rest of this chapter, we will refer to the integer terms of b, b_1, b_2 from Definitions 4.1-4.2 as *symbolic bounds* and the formulas $F[x]$ from the induction axioms of Definitions 4.1-4.2 as *induction target formulas.*

Definition 4.3 (Downward/Upward Induction Rules)**.** The *downward (respectively, upward) induction rule with symbolic bounds*, or simply *downward (respectively, upward)* *induction rule* is the inference rule whose instances are all downward (respectively, upward) induction axioms with symbolic bounds.

Likewise, the *interval downward (respectively, upward) induction rule with symbolic bounds*, or simply *interval downward (respectively, upward) induction rule* is the inference rule whose instances are all interval downward (respectively, upward) induction axioms with symbolic bounds. □

It is easy to see that the new induction rules are sound.

Theorem 4.4 (Soundness)**.** The (interval) downward/upward induction rules of Definition 4.3 are sound, that is, all corresponding induction axioms from Definitions 4.1-4.2 are valid.

Proof. The validity of the induction axioms follows from a straightforward inductive argument. □

4.3 Integer Induction in Saturation-Based Proof Search

Our next aim is to define analogues of the induction rules introduced in Section 4.2 that can be used in superposition theorem provers and their saturation algorithms.

The most general way to introduce our new induction rules at the calculus level is to add clausal forms of our new induction axioms to the search space. That is, for every induction axiom G from Section 4.2, we add the rule

$$\frac{}{G}\ .$$

However, we cannot efficiently implement such a calculus, as any formula $F[x]$ with one variable can be used as an induction target formula. We will therefore introduce different, more specialized rules, which still correspond to the previously defined induction rules. The new rules use variations of the following three ideas:

1. Use only simple induction target formulas, for example literals;
2. To find an induction target formula, generalize a subgoal occurring in the search space. Then the derived induction target formula can be immediately used to prove this subgoal;
3. Use (symbolic) bounds that correspond to bounds already occurring in the search space.

The first two ideas were already used in rules Ind [RV19] and IndGen (see Chapter 3). In particular, given a ground literal $\overline{L}[t]$ in the search space, Ind introduces an induction axiom instantiated with $L[x]$ and thus with $L[x]$ in the conclusion. This $L[x]$ is then resolved against $\overline{L}[t]$.

The third idea is new. Note that, if we use the first two ideas and the upward induction rule, instead of $L[x]$ we will derive $b \leq x \to L[x]$. When we resolve this against $\overline{L}[t]$, we obtain the clause $\neg(b \leq t)$. However, if we already previously derived $b \leq t$, we can also resolve away $\neg(b \leq t)$. This gives us the idea to only apply the upward induction rules when we have $b \leq t$.[1]

Based on the three ideas above, we introduce the following four induction rules on clauses. In these rules t is a ground term, b is a constant and $L[x]$ is a literal containing at least one occurrence of a variable x and no other variables. The rules depend on which comparisons among $t \geq b$, $t > b$, $t \leq b$ and $t < b$ already occur in the current search space:

$$\frac{\overline{L}[t] \vee C \qquad t \geq b}{(L[b] \wedge \forall y.(y \geq b \wedge L[y] \to L[y+1])) \to \forall x.(x \geq b \to L[x])}\ (\mathsf{IntInd}_{\geq})$$

$$\frac{\neg L[t] \vee C \qquad t > b}{(L[b] \wedge \forall y.(y \geq b \wedge L[y] \to L[y+1])) \to \forall x.(x > b \to L[x])}\ (\mathsf{IntInd}_{>})$$

$$\frac{\neg L[t] \vee C \qquad t \leq b}{(L[b] \wedge \forall y.(y \leq b \wedge L[y] \to L[y-1])) \to \forall x.(x \leq b \to L[x])}\ (\mathsf{IntInd}_{\leq})$$

$$\frac{\neg L[t] \vee C \qquad t < b}{(L[b] \wedge \forall y.(y \leq b \wedge L[y] \to L[y-1])) \to \forall x.(x < b \to L[x])}\ (\mathsf{IntInd}_{<})$$

Note that $\mathsf{IntInd}_{\geq}$ and $\mathsf{IntInd}_{>}$ are upward induction rules, whereas $\mathsf{IntInd}_{\leq}$ and $\mathsf{IntInd}_{<}$ are downward induction rules. One can also introduce non-ground analogues of these rules but we do not consider them in this work.

Similarly to the above rules on the clausal level, we also introduce the interval upward/downward induction rules on clauses to be used in saturation algorithms for the superposition calculus. Here we present rule $\mathsf{IntInd}_{[\geq]}$ for interval upward induction, and show the rest of the rules in Figure 4.2. For a ground term t, constants b_1, b_2, and $L[x]$ a literal containing at least one occurrence of a variable x and no other variables, an interval upward induction rule is:

$$\frac{\neg L[t] \vee C \qquad t \geq b_1 \qquad t \leq b_2}{(L[b_1] \wedge \forall y.(b_1 \leq y < b_2 \wedge L[y] \to L[y+1])) \to \forall x.(b_1 \leq x \leq b_2 \to L[x])}\ (\mathsf{IntInd}_{[\geq]})$$

In view of Theorem 4.4, all induction rules of Section 4.2 are sound. Therefore, also all the induction rules of this section are sound. Finally, when we use these rules in saturation, the induction axioms get clausified. Assuming that our clausification function

[1]Using the AVATAR architecture [Vor14], we can easily obtain valid literals $b \leq t$.

$$\frac{\neg L[t] \vee C \qquad t \geq b_1 \qquad t < b_2}{(L[b_1] \wedge \forall y.(b_1 \leq y < b_2 \wedge L[y] \rightarrow L[y+1])) \rightarrow \forall x.(b_1 \leq x < b_2 \rightarrow L[x])} \;(\mathsf{IntInd}_{[\geq']})$$

$$\frac{\neg L[t] \vee C \qquad t > b_1 \qquad t \leq b_2}{(L[b_1] \wedge \forall y.(b_1 \leq y < b_2 \wedge L[y] \rightarrow L[y+1])) \rightarrow \forall x.(b_1 < x \leq b_2 \rightarrow L[x])} \;(\mathsf{IntInd}_{[>]})$$

$$\frac{\neg L[t] \vee C \qquad t > b_1 \qquad t < b_2}{(L[b_1] \wedge \forall y.(b_1 \leq y < b_2 \wedge L[y] \rightarrow L[y+1])) \rightarrow \forall x.(b_1 < x < b_2 \rightarrow L[x])} \;(\mathsf{IntInd}_{[>']})$$

$$\frac{\neg L[t] \vee C \qquad t \leq b_1 \qquad t \geq b_2}{(L[b_1] \wedge \forall y.(b_1 \geq y > b_2 \wedge L[y] \rightarrow L[y-1])) \rightarrow \forall x.(b_1 \geq x \geq b_2 \rightarrow L[x])} \;(\mathsf{IntInd}_{[\leq]})$$

$$\frac{\neg L[t] \vee C \qquad t \leq b_1 \qquad t > b_2}{(L[b_1] \wedge \forall y.(b_1 \geq y > b_2 \wedge L[y] \rightarrow L[y-1])) \rightarrow \forall x.(b_1 \geq x > b_2 \rightarrow L[x])} \;(\mathsf{IntInd}_{[\leq']})$$

$$\frac{\neg L[t] \vee C \qquad t < b_1 \qquad t \geq b_2}{(L[b_1] \wedge \forall y.(b_1 > y > b_2 \wedge L[y] \rightarrow L[y-1])) \rightarrow \forall x.(b_1 > x \geq b_2 \rightarrow L[x])} \;(\mathsf{IntInd}_{[<]})$$

$$\frac{\neg L[t] \vee C \qquad t < b_1 \qquad t > b_2}{(L[b_1] \wedge \forall y.(b_1 > y > b_2 \wedge L[y] \rightarrow L[y-1])) \rightarrow \forall x.(b_1 > x > b_2 \rightarrow L[x])} \;(\mathsf{IntInd}_{[<']})$$

Figure 4.2: Integer interval rules for a ground term t, constants b_1, b_2, and $L[x]$ a literal containing at least one occurrence of a variable x and no other variables.

preserves satisfiability, we conclude that also adding the clausified induction formulas from our rules ($\mathsf{IntInd}_{\geq}$, $\mathsf{IntInd}_{>}$, $\mathsf{IntInd}_{\leq}$, $\mathsf{IntInd}_{<}$, $\mathsf{IntInd}_{[\geq]}$, $\mathsf{IntInd}_{[\geq']}$, $\mathsf{IntInd}_{[>]}$, $\mathsf{IntInd}_{[>']}$, $\mathsf{IntInd}_{[\leq]}$, $\mathsf{IntInd}_{[\leq']}$, $\mathsf{IntInd}_{[<]}$, $\mathsf{IntInd}_{[<']}$) is a sound reasoning step.

Example 4.5. To illustrate again how the choice of induction target formulas allows us to have shorter clauses, consider $\mathsf{IntInd}_{\leq}$. The CNF in its conclusion consists of three clauses:

$$\begin{array}{l} \neg L[b] \vee \sigma \leq b \vee \neg y \leq b \vee L[y] \\ \neg L[b] \vee L[\sigma] \vee \neg y \leq b \vee L[y] \\ \neg L[b] \vee \neg L[\sigma - 1] \vee \neg y \leq b \vee L[y] \end{array} \tag{4.9}$$

These clauses can be resolved against premises of $\mathsf{IntInd}_{\leq}$, yielding the following clauses:

$$\begin{array}{l} \neg L[b] \vee \sigma \leq b \vee C \\ \neg L[b] \vee L[\sigma] \vee C \\ \neg L[b] \vee \neg L[\sigma - 1] \vee C \end{array} \tag{4.10}$$

They have an especially simple form when C is the empty clause $\square$. In this case we have three clauses:

$$\begin{array}{l} \neg L[b] \vee \sigma \leq b \\ \neg L[b] \vee L[\sigma] \\ \neg L[b] \vee \neg L[\sigma - 1] \end{array} \tag{4.11}$$

which subsume the original three longer clauses and are ground. Since they are ground, they can be handled efficiently by Avatar. □

Example 4.6. Let us now demonstrate how the downward induction rule $\mathsf{IntInd}_{\leq}$ works for refuting the inductive property (4.3) from our motivating example of Figure 4.1(a). We use literals from (4.5) as the premises of the $\mathsf{IntInd}_{\leq}$ rule. The corresponding instance of the downward induction rule is given by

$$\begin{aligned} b &:= \sigma_m; \\ t &:= \sigma_n; \\ L[x] &:= 2 \cdot \mathsf{sum}(x, \sigma_m) \simeq \sigma_m \cdot (\sigma_m + 1) - x \cdot (x - 1). \end{aligned}$$

This instance of $\mathsf{IntInd}_{\leq}$ is:

$$\frac{2 \cdot \mathsf{sum}(\sigma_n, \sigma_m) \not\simeq \sigma_m \cdot (\sigma_m + 1) - \sigma_n \cdot (\sigma_n - 1) \qquad \sigma_n \leq \sigma_m}{\begin{aligned}\big(2 \cdot \mathsf{sum}(\sigma_m, \sigma_m) \simeq \sigma_m \cdot (\sigma_m + 1) - \sigma_m \cdot (\sigma_m - 1) \\ \wedge \forall y.(y \leq \sigma_m \to 2 \cdot \mathsf{sum}(y, \sigma_m) \simeq \sigma_m \cdot (\sigma_m + 1) - y \cdot (y - 1) \\ \to 2 \cdot \mathsf{sum}(y - 1, \sigma_m) \simeq \sigma_m \cdot (\sigma_m + 1) - (y - 1) \cdot ((y - 1) - 1))\big) \\ \to \forall x.(x \leq \sigma_m \to 2 \cdot \mathsf{sum}(x, \sigma_m) \simeq \sigma_m \cdot (\sigma_m + 1) - x \cdot (x - 1))\end{aligned}} \ (\mathsf{IntInd}_{\leq})$$

This single instance of the induction rule does the magic. After clausifying the conclusion of the $\mathsf{IntInd}_{\leq}$ rule as in (4.9), we can resolve it against the premises, and obtain the following instances of the clauses from (4.11), where σ is a fresh skolem constant:

$$2 \cdot \mathsf{sum}(\sigma_m, \sigma_m) \not\simeq \sigma_m \cdot (\sigma_m + 1) - \sigma_m \cdot (\sigma_m - 1) \vee \sigma \leq \sigma_m \tag{4.12}$$

$$\begin{aligned} &2 \cdot \mathsf{sum}(\sigma_m, \sigma_m) \not\simeq \sigma_m \cdot (\sigma_m + 1) - \sigma_m \cdot (\sigma_m - 1) \\ &\qquad \vee\ 2 \cdot \mathsf{sum}(\sigma, \sigma_m) \simeq \sigma_m \cdot (\sigma_m + 1) - \sigma \cdot (\sigma - 1) \end{aligned} \tag{4.13}$$

$$\begin{aligned} &2 \cdot \mathsf{sum}(\sigma_m, \sigma_m) \not\simeq \sigma_m \cdot (\sigma_m + 1) - \sigma_m \cdot (\sigma_m - 1) \\ &\qquad \vee\ 2 \cdot \mathsf{sum}(\sigma - 1, \sigma_m) \not\simeq \sigma_m \cdot (\sigma_m + 1) - (\sigma - 1) \cdot ((\sigma - 1) - 1) \end{aligned} \tag{4.14}$$

The first literal of all three clauses (4.12)-(4.14), which is the same for all three clauses, can be refuted by first using the axiom (4.1), and then by simple arithmetic (applied by using superposition on the clauses and on suitable instances of axioms of arithmetic). Thus, the clauses (4.12)-(4.14) are reduced to their latter literals. From the second literal of (4.12) we derive (again by using a suitable axiom of $\leq$) the literal $\sigma - 1 < \sigma_m$. Then we use the following instance of axiom (4.2):

$$\sigma - 1 < \sigma_m \to \mathsf{sum}(\sigma - 1, \sigma_m) \simeq (\sigma - 1) + \mathsf{sum}((\sigma - 1) + 1, \sigma_m) \tag{4.15}$$

We resolve the CNF of (4.15) with $\sigma - 1 < \sigma_m$, and then use the resulting equation, $\mathsf{sum}(\sigma - 1, \sigma_m) \simeq (\sigma - 1) + \mathsf{sum}((\sigma - 1) + 1, \sigma_m)$, together with the second literal of (4.14), and derive:

$$2 \cdot ((\sigma - 1) + \mathsf{sum}((\sigma - 1) + 1, \sigma_m)) \not\simeq \sigma_m \cdot (\sigma_m + 1) - (\sigma - 1) \cdot ((\sigma - 1) - 1) \tag{4.16}$$

By simple arithmetic together with (4.16), we obtain:

$$2 \cdot \mathsf{sum}(\sigma, \sigma_m) \not\simeq \sigma_m \cdot (\sigma_m + 1) - \sigma \cdot (\sigma - 1) \tag{4.17}$$

Ultimately, we resolve (4.17) with the second literal of (4.13) and obtain □, which concludes the refutation of (4.5) and therefore also the proof of (4.3). The proof of the problem is also evidenced by the results for the first problem subset, *x_all* of *sum*, in Table 6.5 of Chapter 6.

We finally note that functional correctness of Figure 4.1(b) is proved by the interval upward induction rule $\texttt{IntInd}_{[\geq]}$, in a similar way as above, and as evidenced by the results of Table 6.5 for *declared_unint_ax-fin_conj-fin* in *val*. □

What we find especially interesting in Example 4.6 is that the induction axiom used in it (and discovered by our implementation of induction in Vampire) uses the induction argument that would probably be used by a majority of humans who would try to argue why the program property holds.

CHAPTER 5

Inductive Benchmarks

The contributions of this chapter are based on:
Márton Hajdu, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Inductive Benchmarks for Automated Reasoning. In Fairouz Kamareddine and Claudio Sacerdoti Coen, editors, Proc. of CICM*, volume 12833 of* LNCS*, pages 124–129, Cham, 2021. Springer [HHK+21]*

Evaluation of our developments from Chapters 3 and 4 prompts comparison not only among first-order theorem provers [Cru17] and/or SMT solvers [RK15], but also with inductive provers (e.g., ACL2 [BM79], Zeno [SDE12] or Imandra [PCI+20]). In this chapter we describe a benchmark set of 3516 benchmarks based on variations of properties of inductive data types as well as integers, which we created as a part of our work on automating induction in Vampire. We primarily present our benchmarks in the SMT-LIB input format [BFT16]. To facilitate comparison of different solvers and provers, we also provide translations of the benchmarks into the input formats of other state-of-the-art inductive reasoners, for example as functional program encodings.

Our benchmark set is available at:

`https://github.com/vprover/inductive_benchmarks`

5.1 Benchmark Format

We provide all benchmarks in the standard SMT-LIB 2.6 syntax. We chose SMT-LIB as the main format for our benchmarks, since it is the most common format used by automated reasoners (SMT solvers and first-order provers, e.g., cvc5 [BBB+22] or Vampire [KV13]) and verification tools (e.g., CBMC [KT14], Dafny [Lei10], or eThor [SGSM20]). In our examples, we use the SMT-LIB construct `declare-fun` to declare functions and `assert` to axiomatize functions (see the example benchmarks in Section 5.2). In addition to the SMT-LIB syntax, we also translated our examples to other formats depending on the data types used in these examples: three subsets of our benchmark set use inductively defined data types, and one subset uses integers (see Section 5.2). For the benchmarks with inductively defined data types, we also provide SMT-LIB encoding using the `define-fun-rec` construct for recursive function definitions.

Besides the SMT-LIB format, we also provide our benchmarks translated into other, less common input formats supported by state-of-the-art solvers for automating induction. Namely, for our benchmarks with inductively defined data types, we provide two encodings for Zipperposition [Cru17] (using Zipperposition's native input format `.zf` with/without function definitions encoded as rewrite rules), and when possible[1] functional program encodings for ACL2 [BM79] (in Lisp), Imandra [PCI+20] (in OCaml) and Zeno [SDE12] (in Haskell). Since the solvers support a broad variety of logics, the benchmarks must be encoded in a logic belonging to a common subset. Further, since the input formats of Zeno and Imandra only support properties of functional programs, the axioms of $+_{\mathbb{N}}$ and $+\!\!+$ were translated to function definitions by pattern matching, while the predicates $\leq_{\mathbb{N}}$ and pref were translated into boolean functions.[2]

For our inductive benchmarks over integers, we only provide a translation into Lisp for ACL2. To the best of our knowledge, in addition to Vampire [HKV21] and cvc5 [RK15], ACL2 is the only prover supporting inductive reasoning with integers.

5.2 Benchmark Categories

Our benchmark set consists of two categories, requiring different kinds of inductive reasoning, as follows. The benchmark category `dty` uses structural induction over inductively defined data types, whereas our `int` benchmark suite exploits integer induction.

To confirm that our new benchmarks require the use of inductive reasoning, we tested them on the SMT solver Z3 [DMB08] that does not support induction. Z3 solved 17 out of the 3396 problems from the `dty` set, and could not solve any of the 120 problems from the `int` set.

5.2.1 `dty` - Benchmarks with Inductively Defined Data Types

The 3396 problems within the category `dty` involve three different inductively defined data types: natural numbers $\mathbb{N}$, lists of natural numbers $\mathbb{L}$, and binary trees of natural

[1]Some concepts, like conjectures that contain existential quantification, or some uninterpreted functions used to model out-of-bounds access for list indexing, are not straightforwardly translatable into these formats.

[2]This is why we defined $\leq_{\mathbb{N}}$ the way we did in Figure 2.1 instead of using a more common definition, such as $\{\forall x. x \leq_{\mathbb{N}} x, \forall x. \forall y. (x \leq_{\mathbb{N}} y \rightarrow x \leq_{\mathbb{N}} \mathsf{s}(y))\}$.

numbers $\mathbb{BT}$. These data types are defined as follows:

```
(declare-datatypes ((nat 0) (list 0) (tree 0))
  (((zero) (s (s0 nat)))
   ((nil) (cons (head nat) (tail list)))
   ((Nil) (node (lc tree) (val nat) (rc tree)))))
```

The set is split into three subcategories `nat`, `list`, and `tree`, depending on the algebraic data types used in the examples. The category `nat` uses natural numbers only, `list` uses lists and natural numbers, and `tree` uses all three of the data types. Each of these categories within `dty` contains examples defining functions and predicates on the respective data type and a conjecture/goal to prove about these functions and predicates, as described next. To avoid repetition in the displayed examples, we use short descriptions of repeated content beginning with the comment sign `;-`.

Part of the benchmarks are *hand-crafted* based on natural mathematical problems. Some examples were taken from or inspired by the TIP benchmark library [CJRS15]. An example benchmark of the `dty` set is the problem of associativity of addition with only one variable (3.4) which we recall here:

$$\forall x \in \mathbb{N}.\ x +_{\mathbb{N}} (x +_{\mathbb{N}} x) \simeq (x +_{\mathbb{N}} x) +_{\mathbb{N}} x$$

This is a special case of a family of problems over natural numbers. The problems can be formulated as follows:

Let t_1 and t_2 be two terms built using variables, $+_{\mathbb{N}}$ and the successor function. Then the equality $t_1 \simeq t_2$ is valid over natural numbers if and only if they have the same number of occurrences of the successor function and each variable of this equality has the same number of occurrences in t_1 and t_2.

For example, the following equality is valid:

$$\forall x, y, z \in \mathbb{N}.\ \mathsf{s}(x +_{\mathbb{N}} (x +_{\mathbb{N}} \mathsf{s}(y +_{\mathbb{N}} z))) +_{\mathbb{N}} \mathsf{s}(z) \simeq (z +_{\mathbb{N}} \mathsf{s}(x)) +_{\mathbb{N}} (x +_{\mathbb{N}} \mathsf{s}(\mathsf{s}((z +_{\mathbb{N}} y))))$$

To prove such problems over natural numbers, one needs both induction and generalization. Without the successor function, they can be easily proved using associativity and commutativity of $+_{\mathbb{N}}$, but associativity and commutativity are not included in the axioms of $\mathbb{N}$. When the terms are large, the problems become highly challenging.

We generated a set of instances of these problems (with and without the successor function, and also other functions and predicates) by increasing term sizes. We also generated similar problems for lists using the concatenation and reverse functions, and the prefix predicate. Some of the terms were, e.g., variations of (3.4) with 20 occurrences of x. These *generated* instances of various sizes form the second part of the `dty` benchmark set.

The `dty` benchmarks were used for evaluating the work from Chapter 3 (see Section 6.1).

nat Examples. The category nat contains a set of hand-crafted benchmarks encoding basic properties of natural numbers like commutativity of addition and multiplication. Additionally, nat contains three groups of generated benchmarks. In group add_<m>var_<n>occ, the conjecture of each benchmark consists of an equality of two sums of variables, with arbitrary bracketing, and n variables on each side of the equality, where m distinct variables occur in the conjecture. In group add_<n>sym, the conjectures are equalities with an arbitrary combination of the successor function, zero, addition, and variables, on both hand sides. Each side of the equality in these benchmarks contains n symbols in total. The group leq_<m>var_<n>_<o>occ has a less-or-equal inequality as conjecture. It contains m distinct variables, with a total of n variables on the left-hand side arbitrarily added up, and a total of o variables occurring on the right-hand side, where each variable on the left-hand side is contained on the right-hand side at least as often as on the left one in order to ensure that the conjecture is indeed valid.

Inductive nat example from the set add_2var_4occ

```
(set-logic UFDT)

(declare-datatypes ((nat 0)) (((zero) (s (s0 nat)))))

(declare-fun add (nat nat) nat)
(assert (forall ((y nat)        ) (= (add  zero y) y            )))
(assert (forall ((x nat) (y nat)) (= (add (s x) y) (s (add x y)))))

(assert (not (forall ((v0 nat) (v1 nat))
  (= (add (add v0 (add v1 v1)) v1) (add (add (add v1 v1) v1) v0)))))

(check-sat)
```

The conjecture is a combination of associativity and commutativity of addition of natural numbers for two variables with four occurrences in total:

$$\forall v_0, v_1 \in \mathbb{N}.\ (v_0 +_{\mathbb{N}} (v_1 +_{\mathbb{N}} v_1)) +_{\mathbb{N}} v_1 \simeq ((v_1 +_{\mathbb{N}} v_1) +_{\mathbb{N}} v_1) +_{\mathbb{N}} v_0$$

list Examples. These examples describe basic properties of lists, such as relating concatenation of lists to the resulting list length. Similarly to nat, the category list also contains two generated example sets: concat_<m>var_<n>occ contains examples as in add_<m>var_<n> occurrences, but using list concatenation instead of list addition, while pref_<m>var_<n>_<o>occ is defined in the same way as leq_<m>var_<n>_<o>occ, but replacing the less-or-equal order with the prefix relation and using list concatenation instead of natural addition.

Inductive list example from the set crafted

```
(set-logic UFDT)

(declare-datatypes ((nat 0) (list 0) (tree 0))
  (((zero) (s (s0 nat)))
   ((nil) (cons (head nat) (tail list)))))
```

```
;- add function declaration & axiomatization, as in the example above

(declare-fun app (list list) list)
(assert (forall ((r list) ) (= (app nil r) r)))
(assert (forall ((a nat) (l list) (r list))
  (= (app (cons a l) r) (cons a (app l r)))))
(declare-fun len (list) nat)
(assert                              (= (len nil       ) zero        ))
(assert (forall ((e nat) (l list)) (= (len (cons e l)) (s (len l)))))

(assert (not (forall ((x list) (y list))
  (= (add (len x) (len y)) (len (app x y))))))

(check-sat)
```

The conjecture asserts that addition of lengths of two lists is equal to the length of the two lists concatenated:

$$\forall x, y \in \mathbb{L}.\ \mathsf{len}(x) +_{\mathbb{N}} \mathsf{len}(y) \simeq \mathsf{len}(x +\!\!+\, y)$$

`tree` Examples. This category consists only of hand-crafted benchmarks, and has two main subcategories: one problem set relates binary trees indirectly by flattening them to lists, the other relates them directly to each other. The defined functions are two in-order flattening variants, two functions that recursively rotate a tree completely to the left and to the right at its root, one counting the number of non-leaf nodes in a tree and one checking if two trees are mirror images of each other. Occurrences of the flattening and rotating functions are varied to get variants for each problem.

Inductive `tree` example from the set `flatten0_rotate_5var`

```
(set-logic UFDT)

(declare-datatypes ((nat 0) (list 0) (tree 0))
  (((zero) (s (s0 nat)))
   ((nil) (cons (head nat) (tail list)))
   ((Nil) (node (lc tree) (val nat) (rc tree)))))

;- app function declaration & axiomatization, as in the example above

(declare-fun flat0 (tree) list)
(assert (= (flat0 Nil) nil))
(assert (forall ((p tree) (x nat) (q tree))
  (= (flat0 (node p x q)) (app (flat0 p) (cons x (flat0 q))))))

(assert (not (forall ((p tree) (q tree) (r tree) (x nat) (y nat))
  (= (flat0 (node (node p x q) y r)) (flat0 (node p x (node q y r))))
)))

(check-sat)
```

$\underline{\textbf{assume}}\ \ e \geq_{\mathbb{Z}} 1$

$\textbf{fun}\ \mathsf{power}(x, 1) = x$
$\quad \mid \mathsf{power}(x, e) = x \cdot_{\mathbb{Z}} \mathsf{power}(x, e -_{\mathbb{Z}} 1);$

$\underline{\textbf{assert}}\ \ \forall x, y \in \mathbb{Z}.(\mathsf{power}(x \cdot_{\mathbb{Z}} y, e) \simeq \mathsf{power}(x, e) \cdot_{\mathbb{Z}} \mathsf{power}(y, e))$

Figure 5.1: ML-like functional program computing integer powers for positive exponents.

The conjecture asserts that the result of a tree flattening does not depend on the rotation in the root:

$$\forall p, q, r \in \mathbb{BT}.\ \forall x, y \in \mathbb{N}.\ \mathsf{flat}(\mathsf{node}(\mathsf{node}(p, x, q), y, r)) \simeq \mathsf{flat}(\mathsf{node}(p, x, (\mathsf{node}(q, y, r))))$$

5.2.2 int - Benchmarks with Integers

The `int` category of our benchmark set contains 120 problems and to the best of our knowledge it is the first published benchmark set focused on inductive reasoning with integers. It is inspired by software verification problems for three programs:

1. `power`, computing powers of integers. We show an example program in Figure 5.1. A corresponding encoding in first-order logic is:

$$\begin{aligned} \text{axioms:}\quad & \forall x \in \mathbb{Z}.\ \mathsf{power}(x, 1) \simeq x \\ & \forall x, e \in \mathbb{Z}.\ (2 \leq_{\mathbb{Z}} e \rightarrow \mathsf{power}(x, e) \simeq x \cdot_{\mathbb{Z}} \mathsf{power}(x, e -_{\mathbb{Z}} 1)) \\ \text{conjecture:}\quad & \forall x, y, e \in \mathbb{Z}.(1 \leq_{\mathbb{Z}} e \rightarrow \mathsf{power}(x \cdot_{\mathbb{Z}} y, e) \simeq \mathsf{power}(x, e) \cdot_{\mathbb{Z}} \mathsf{power}(y, e)) \end{aligned} \tag{5.1}$$

2. `sum`, computing sums of integer intervals. We show an example program in Figure 4.1(a). The corresponding first-order encoding forms the axioms (4.1-4.2) and the assertion (4.3).

3. `val`, using integers as array indices to encode array properties. We show an example program in Figure 4.1(b). We encode the invariant and the negation of the loop condition as the axioms (4.7), and the assertion as (4.6).

A sample problem from `power` corresponding to the program in Figure 5.1 and encoding (5.1), expressing that the recursively defined power function on integers for positive exponents is distributive over multiplication, is:

Inductive `int` example from the set `power`

```
(set-logic UFNIA)

(declare-fun pow (Int Int) Int)
(assert (forall ((x Int)) (= (pow x 1) x)))
(assert (forall ((x Int) (e Int))
```

Set	Variant tag	Description
sum	*x / y*	$\mathsf{sum}(x, y)$ for $x >_{\mathbb{Z}} y$ defined as $x +_{\mathbb{Z}} \mathsf{sum}(x +_{\mathbb{Z}} 1, y)$ or $y +_{\mathbb{Z}} \mathsf{sum}(x, y -_{\mathbb{Z}} 1)$
	all / geq / leq	the conjecture holds for all x, y where $x \leq_{\mathbb{Z}} y$, or only for $x \leq_{\mathbb{Z}} y \simeq c$, or only for $c \simeq x \leq_{\mathbb{Z}} y$; where $c \in \mathbb{Z}$ is an interpreted constant
val	*declared / defined*	val was either not defined, only declared and axiomatized (as in (4.6)), or defined as a total computable function (as in (5.3))
	inter / unint / mixed	the axiom and conjecture use concrete interpreted constants, or uninterpreted constants, or a mix of both
	ax-fin / ax-all / ax-leq / ax-geq	the axiom holds for integers in an interval $[c, c')$, or for all $x \in \mathbb{Z}$, or only for $x \leq_{\mathbb{Z}} c$, or only for $x \geq_{\mathbb{Z}} c$; where $c, c' \in \mathbb{Z}$ are constants
	conj-fin / conj-all / conj-leq / conj-geq	the conjecture holds for integers in an interval $[c, c']$, or for all integers, or only for integers $\leq_{\mathbb{Z}} c$, or only for integers $\geq_{\mathbb{Z}} c$; where $c, c' \in \mathbb{Z}$ are constants
power	*0 / 1*	power defined starting with $\mathsf{power}(x, 0) \simeq 1$ or $\mathsf{power}(x, 1) \simeq x$
	all / pos / neg	the conjecture holds either for all x, y, or only for $x, y \geq_{\mathbb{Z}} 0$, or only for $x, y \leq_{\mathbb{Z}} 0$

Table 5.1: Description of the `int` benchmark set.

```
  (=> (<= 2 e) (= (pow x e) (* x (pow x (- e 1)))))))

(assert (not (forall ((x Int) (y Int) (e Int))
  (=> (<= 1 e) (= (pow (* x y) e) (* (pow x e) (pow y e)))))))

(check-sat)
```

The example corresponds to the program in Figure 5.1 and its encoding from (5.1).

All variations of the `int` benchmarks were created by varying the constraints and constants in the definitions and goals as described in Table 5.1. For example, variations of the sample problem above use the function power defined starting from 0 instead of 1, or introduce additional constraints on variables x, y, and e in the conjecture.

Names of subsets of our new benchmarks are constructed by joining variant tags described in Table 5.1. For example, problem (4.6) belongs to the category *declared_unint_ax-fin_conj-fin* of the set *val*. The following benchmark:

$$\begin{aligned} \text{axiom:} \quad & \forall x \in \mathbb{Z}.\ \mathsf{val}_{\mathrm{A}}(x) \simeq \mathsf{val}_{\mathrm{A}}(x +_{\mathbb{Z}} 1) \\ \text{conjecture:} \quad & \forall x, y \in \mathbb{Z}.\ \mathsf{val}_{\mathrm{A}}(x) \simeq \mathsf{val}_{\mathrm{A}}(y) \end{aligned} \tag{5.2}$$

belongs to *declared_unint_ax-all_conj-all* of *val* and the below example is from *defined_inter_ax-geq_conj-geq* of *val*:

$$\begin{aligned} \text{axioms:} \quad & \forall x \in \mathbb{Z}.(x \leq_{\mathbb{Z}} 0 \rightarrow \mathsf{val}_{\mathrm{A}}(x) \simeq 0) \\ & \forall x \in \mathbb{Z}.(0 <_{\mathbb{Z}} x \rightarrow \mathsf{val}_{\mathrm{A}}(x) \simeq \mathsf{val}_{\mathrm{A}}(x -_{\mathbb{Z}} 1)) \\ \text{conjecture:} \quad & \forall x \in \mathbb{Z}.(0 \leq_{\mathbb{Z}} x \rightarrow \mathsf{val}_{\mathrm{A}}(x) \simeq \mathsf{val}_{\mathrm{A}}(0)) \end{aligned} \tag{5.3}$$

While 9 of the benchmarks (all in *val*) use finite intervals in both the assertion and the invariant (*ax-fin_conj-fin*), the remaining 111 benchmarks require inductive reasoning over infinite intervals.

The benchmarks from this set were used for evaluating the work from Chapter 4 (see Section 6.2).

CHAPTER 6

Implementation of Induction in Vampire

The contributions of this chapter are based on:

Márton Hajdu, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Induction with Generalization in Superposition Reasoning. In Christoph Benzmüller and Bruce Miller, editors, Proc. of CICM, *volume 12236 of* LNCS, *pages 123–137, Cham, 2020. Springer [HHK+20],*

Petra Hozzová, Laura Kovács, and Andrei Voronkov. Integer Induction in Saturation. In André Platzer and Geoff Sutcliffe, editors, Proc. of CADE, *volume 12699 of* LNCS, *pages 361–377, Cham, 2021. Springer [HKV21]*

We describe the implementation and experimental evaluation of the methods from chapters 3 and 4. We implemented the methods in the first-order theorem prover Vampire [KV13]. We differentiate between different versions of Vampire by using subscripts. In particular, by Vampire$_o$ we denote the original version of Vampire without our new developments. Our implementation is available online at:

`https://github.com/vprover/vampire`

We note that in this chapter we compare the performance of our implementation with the solver CVC4 [BCD+11], and not its newer version cvc5 [BBB+22]. Our initial experiments were performed in 2020-2021 with the most recent version of the solver that was available at that time. The inductive reasoning features are also supported in cvc5 [BBB+22] in a similar way as in CVC4.

6.1 Induction with Generalization

We implemented the method from Chapter 3 in Vampire and compared it to state-of-the-art reasoners automating induction, including ACL2 [BM79, KMM00], CVC4 [BCD+11], Imandra [PCI+20], Zeno [SDE12] and Zipperposition [Cru17] on benchmarks described in Subsection 5.2.1. We show that induction with generalization in Vampire can solve problems that existing systems, including Vampire without this rule, cannot.

6.1.1 Implementation

We implemented induction with generalization in Vampire. Our implementation consists of approximately 350 lines of C++ code additional to Vampire's pre-existing induction features. We introduced two new options:

- boolean-valued option `indgen`, which turns on/off the application of induction with generalization, with the default value being off, and
- integer-valued option `indgenss`, which sets the maximum size of the subset of occurrences used for induction, with the default value 3. This option is ignored if `indgen` is off.

In experiments described here, if `indgen` is off, Vampire performs induction on all occurrences of a term in a literal as in [RV19]. In this section

- Vampire$_o$ refers to the (default) version of Vampire with induction rule Ind (2.1) (i.e., the option `-ind struct`).
- Vampire$_g$ additionally uses the IndGen rule of induction with generalization (3.8) (i.e., the options `-ind struct -indgen on`).
- Vampire$_{gc}$ uses the same options as Vampire$_g$ plus the option `-indoct on`, which applies induction to arbitrary ground terms, not just to constants as in Vampire$_o$ or in Vampire$_g$.

6.1.2 SMT-LIB experiments

We evaluated our work using the UFDT and UFDTLIA problem sets from SMT-LIB [BFT16], yielding all together 4854 problems. Many of these problems come from program analysis and verification and contain large numbers of axioms, so they are different from standard mathematical examples used in many other papers on automation of induction. Given the nature of the benchmarks, we were interested in two questions:

1. What is the overhead incurred by using induction with generalization in large search spaces, especially when it is not used in proofs? If the new rule is prohibitively

expensive, this means it could probably only be used in smaller examples used in interactive theorem proving.

2. Is the new rule useful at all for this kind of benchmarks? While the new rule can be used in principle, should it (or can it) be used in program analysis and verification?

Our results show that *the overhead is relatively small* but we could not solve problems not solvable without the use of the new rule.

Induction rule Ind (2.1) in Vampire$_o$ was already evaluated in [RV19] against other solvers on these examples. Hence, we only compare how Vampire$_g$/Vampire$_{gc}$ performs against Vampire$_o$, using both the default and the portfolio modes. (In the default mode, Vampire$_o$/Vampire$_g$/Vampire$_{gc}$ uses default values for all parameters except the ones specified by the user; in the portfolio mode, Vampire$_o$/Vampire$_g$/Vampire$_{gc}$ sequentially tries different configurations for parameters not specified by the user.) Together, we ran 18 instances: Vampire$_o$, Vampire$_g$ with `indgenss` set to 2, 3, 4, and unlimited, and Vampire$_{gc}$ with the same four variants of `indgenss`; each of them in both default and portfolio mode. We ran our experiments on the StarExec cluster [SST14].

The best Vampire$_g$/Vampire$_{gc}$ solved 5 problems in the portfolio mode and 1 problem in the default mode not solved by Vampire$_o$. However, the proofs found by them did not use induction with generalization. This is a common problem in experiments with saturation theorem proving: new rules change the direction of the proof search and may result in new simplifications that also drastically affect the search space. As a result, new proofs may be found, yet these proofs do not actually use the new rule. There were no problems solved by Vampire$_o$ that were not solved by any Vampire$_g$/Vampire$_{gc}$.

The maximum number of `IndGen` applications in proofs was 3 and the maximum depth of induction was 4. Vampire$_g$/Vampire$_{gc}$ used generalized induction in proofs of 10 problems. However, these problems are also solvable by Vampire$_o$ (without generalized induction). Thus, we conclude that SMT-LIB problems (probably as well as other typical program analysis and verification benchmarks) typically do not gain from using generalization. However, such examples would typically arise in mathematical properties over naturals/lists, as discussed next.

6.1.3 Experiments with mathematical problems

In this subsection we zoom in on selected hand-crafted benchmarks over natural numbers and lists (see categories `nat` and `list` from Section 5.2). Table 6.1 lists 16 of such examples using the functions defined in Figure 2.1. While the examples are hand-crafted, we believe they are representative of problems requiring generalization, since no attempt was made to exclude problems not solvable by Vampire using induction with generalization in practice.

We evaluated and compared several state-of-the-art reasoners supporting standard input formats and, due to the nature of our work, either superposition-based approaches or

	Theory	Vampire$_g$	Vampire$_{gc}$	Vampire$_o$	Cvc4	Zipperposition	Zeno	Imandra	Acl2	Cvc4-Gen	ZipRewrite
$\forall x, y.(x + y \simeq y + x)$	$\mathbb{N}$	✓	✓	✓	✓	✓	✓	✓	–	✓	✓
$\forall x.(x + \mathsf{s}(x) \simeq \mathsf{s}(x + x))$	$\mathbb{N}$	✓	✓	–	–	–	–	–	–	✓	✓
$\forall x, y, z.(x + (y + z) \simeq (x + y) + z)$	$\mathbb{N}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall x.(x + (x + x) \simeq (x + x) + x)$	$\mathbb{N}$	✓	✓	–	–	–	✓	–	–	✓	✓
$\forall x.((x + x) + ((x + x) + x) \simeq x + (x + ((x + x) + x)))$	$\mathbb{N}$	✓	✓	–	–	–	✓	–	–	✓	✓
$\forall x, y.(y + (x + x) \simeq (x + y) + x)$	$\mathbb{N}$	✓	✓	–	–	–	–	–	–	–	✓
$\forall x.(x \leq x)$	$\mathbb{N}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall x, y.(x \leq x + y)$	$\mathbb{N}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall x.(x \leq x + x)$	$\mathbb{N}$	✓	✓	–	–	–	–	–	–	–	–
$\forall x.(x + x \leq (x + x) + x)$	$\mathbb{N}$	✓	✓	–	–	–	✓	–	–	–	–
$\forall l, k, j.(l +\!\!+ (k +\!\!+ j) \simeq (l +\!\!+ k) +\!\!+ j)$	$\mathbb{L}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall l.(l +\!\!+ (l +\!\!+ l) \simeq (l +\!\!+ l) +\!\!+ l)$	$\mathbb{L}$	✓	✓	–	–	–	–	–	–	–	✓
$\forall l, k.(l +\!\!+ (k +\!\!+ (l +\!\!+ l)) \simeq (l +\!\!+ k) +\!\!+ (l +\!\!+ l))$	$\mathbb{L}$	✓	✓	–	–	–	–	–	–	–	✓
$\forall l, k.\mathsf{pref}(l, l +\!\!+ k)$	$\mathbb{L}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall l.\mathsf{pref}(l, l +\!\!+ l)$	$\mathbb{L}$	✓	✓	–	–	–	–	–	–	–	–
$\forall l : \mathbb{L}, x : \mathbb{N}.(\mathsf{cons}(x+\mathsf{s}(x), l) +\!\!+ (l +\!\!+ l) \simeq (\mathsf{cons}(\mathsf{s}(x) + x, l) +\!\!+ l) +\!\!+ l)$	$\mathbb{N}, \mathbb{L}$	✓	✓	–	–	–	–	–	–	–	–

Table 6.1: Experiments with 16 hand-crafted benchmarks. "✓" denotes success, "–" denotes failure. In this table, we use $+$ to denote $+_{\mathbb{N}}$ and $\leq$ to denote $\leq_{\mathbb{N}}$.

approaches to generalization. It was not easy to carry out these experiments since provers use different input syntaxes (see Table 6.2). As a result, we also had to design translations of our benchmarks.

Except for Imandra (which is a cloud-based service), we ran our experiments on a 2,9 GHz Quad-Core Intel Core i7 machine. We ran each solver as a single-threaded process with a 5-second time limit.

Both Zipperposition and CVC4 feature additional methods for inductive reasoning. Zipperposition implements heuristics for generalizing inductive goals, that are used by default if functions are defined as rewrite rules in functional programming style instead of being defined by ordinary formulas [Cru17]. We refer to Zipperposition run on inputs with functions defined as rewrite rules as ZipRewrite. The additional technique CVC4 features is automatic lemma discovery [RK15]. We refer to CVC4 with this feature as Cvc4-Gen. Zeno supports some heuristics for generalization out of the box.

Configurations used for running all solvers are listed in Table 6.2, and the comparison results are summarized in Table 6.1. All solvers can prove general properties, such as associativity, when it is supplied as a goal. However, most solvers fail to prove instances of these properties. Interestingly, Cvc4-Gen's heuristic discovered associativity of addition, but not its list counterpart for concatenation. ZipRewrite's heuristic works on both plus and concatenation, but cannot handle ordering relations. Zeno's generalization heuristic does not permit for a straightforward interpretation, since it

Solver	Configuration	Input format
VAMPIRE$_o$	`--induction struct`	SMT-LIB
VAMPIRE$_g$	`--induction struct` `--induction_gen on`	SMT-LIB
VAMPIRE$_{gc}$	`--induction struct` `--induction_gen on` `--induction_on_\` `complex_terms on`	SMT-LIB
CVC4	`--quant-ind`	SMT-LIB
CVC4-GEN	`--quant-ind` `--conjecture-gen`	SMT-LIB
ZIPPERPOSITION	default mode	`.zf` (native input format)
ZIPREWRITE	default mode	`.zf` with definitions as rewrite rules
ZENO	default mode	functional program encoding in Haskell
IMANDRA	default mode	functional program encoding in OCaml
ACL2	default mode	functional program encoding in Lisp

Table 6.2: Configurations and input format of solvers for the mathematical problems.

fails for commutativity, but succeeds for associativity. Further, it fails to generalize over variables (row 6), but solves a similar problem by generalizing over terms (row 7). Table 6.1 shows that VAMPIRE$_g$/VAMPIRE$_{gc}$ (with `-indgenss 3`) outperforms all solvers, including VAMPIRE$_o$ itself. When considering solvers without fine-tuned heuristics, such as in ZIPREWRITE and CVC4-GEN, VAMPIRE$_{gc}$ solves many more problems. We believe our experiments show the potential of using induction with generalization as a new inference rule since it outperforms heuristic-driven approaches with no special heuristics or fine-tuning added to VAMPIRE.

6.1.4 Experiments with problems requiring associativity and commutativity

In this subsection we focus on evaluating and comparing various reasoners and approaches on the generated subsets of the benchmark categories `nat` and `list` (see Section 5.2). The interesting feature of these problems is that they are natural yet we can generate problems of almost arbitrary complexity.

We evaluated and compared VAMPIRE$_g$, VAMPIRE$_{gc}$, CVC4-GEN, ZENO and ZIPREWRITE, that is the best performing solvers on inductive reasoning with generalization according to Table 6.1, using the same experimental setting as already described for Table 6.1. Table 6.3 lists a partial summary of our experiments, displaying results for 2,007 large instances of four simple properties with one variable, corresponding to the fourth, ninth, twelfth, and fifteenth problem from Table 6.1. Since the solvers' performance was very similar for the whole benchmark set, we chose these problems as a representative subset of our large benchmarks.

In Table 6.3, we use the following notation. By $nx \simeq nx$ we denote formulas of the form $x \circ \cdots \circ x \simeq x \circ \cdots \circ x$ with n occurrences of x on both sides of the equality, and parentheses in various places in the expressions, with $\circ$ being $+_{\mathbb{N}}$, or $+\!\!+$ for the data types $\mathbb{N}$ and $\mathbb{L}$, respectively. By $mx \leq_{\mathbb{N}} nx$ and $\mathsf{pref}(mx, nx)$ we denote formulas of the form $x +_{\mathbb{N}} \ldots +_{\mathbb{N}} x \leq x +_{\mathbb{N}} \ldots +_{\mathbb{N}} x$ and $\mathsf{pref}(x +\!\!+ \cdots +\!\!+ x, x +\!\!+ \cdots +\!\!+ x)$, respectively, with m occurrences of x on the left and n occurrences of x on the right hand side of the $\leq_{\mathbb{N}}$ or pref predicates, and with parentheses on various places in the expressions. Result $N\%(M)$ means that the solver solved M of the problems from this category, which corresponds to $N\%$.

From Table 6.3, we conclude that Vampire$_{gc}$ scales better than Cvc4-Gen on a large majority of benchmarks, and scales comparably to Zeno. While ZipRewrite can solve more problems than Vampire$_{gc}$, Vampire$_{gc}$ is more consistent in solving at least some problems from each category. ZipRewrite can solve many problems thanks to the treatment of equalities as rewrite rules. We note that Vampire was also later extended by new rules using recursive definitions for rewriting (see [HHKV21]).

6.2 Integer Induction

We implemented the integer induction method from Chapter 4 in Vampire and compared our implementation with other relevant provers, including Vampire without integer induction. Our experiments show that integer induction can completely automatically solve many new problems that could not so far be solved by any prover. For example, 75 problems coming from program analysis and/or mathematical integer properties could be solved only by Vampire with the new induction rules.

To the best of our knowledge, the state-of-the-art systems implementing inductive reasoning have so far not yet considered inductive reasoning over integers, with two exceptions: CVC4 [RK15], which mainly focuses on induction over inductively defined data types but mentions induction on non-negative integers, and ACL2 [KMM00], which supports inductive reasoning using recursive function definitions without any special treatment for integers.

6.2.1 Implementation

We implemented all our integer induction rules from Section 4.3 ($\mathsf{IntInd}_{\geq}$, $\mathsf{IntInd}_{>}$, $\mathsf{IntInd}_{\leq}$, $\mathsf{IntInd}_{<}$, $\mathsf{IntInd}_{[\geq]}$, $\mathsf{IntInd}_{[\geq']}$, $\mathsf{IntInd}_{[>]}$, $\mathsf{IntInd}_{[>']}$, $\mathsf{IntInd}_{[\leq]}$, $\mathsf{IntInd}_{[\leq']}$, $\mathsf{IntInd}_{[<]}$, $\mathsf{IntInd}_{[<']}$) in Vampire. Further, we also implemented a more general induction rule IntInd that does not require bounds to be in the search space and uses 0 as the lower or the upper bound. Our implementation in Vampire consists of approximately 1,200 lines of new C++ code. The size of this additional code is relatively small because Vampire has libraries for indexing and chaining inference rules that could be used off the shelf.

Our (interval) downward/upward induction rules described in Section 4.3 can be applied when either (i) the comparison literal (e.g., $t \geq_{\mathbb{Z}} b$ for the $\mathsf{IntInd}_{\geq}$ rule) is selected and the

	Theory	Vampire$_g$	Vampire$_{gc}$	Cvc4-Gen	Zeno	ZipRewrite
$3x \simeq 3x$	$\mathbb{N}$	100% (1)	100% (1)	100% (1)	100% (1)	100% (1)
$4x \simeq 4x$		90% (9)	100% (10)	100% (10)	20% (2)	100% (10)
$5x \simeq 5x$		30% (15)	50% (25)	100% (50)	12% (6)	100% (50)
$6x \simeq 6x$		8% (4)	18% (9)	100% (50)	22% (11)	100% (50)
$7x \simeq 7x$		–	10% (5)	100% (50)	2% (1)	100% (50)
$8x \simeq 8x$		–	2% (1)	100% (50)	4% (2)	100% (50)
$9x \simeq 9x$		–	2% (1)	100% (50)	8% (4)	84% (42)
$10x \simeq 10x$		–	–	100% (50)	8% (4)	90% (45)
$3x \simeq 3x$	$\mathbb{L}$	100% (1)	100% (1)	–	–	100% (1)
$4x \simeq 4x$		70% (7)	90% (9)	–	–	100% (10)
$5x \simeq 5x$		46% (23)	48% (24)	–	–	100% (50)
$6x \simeq 6x$		6% (3)	26% (13)	–	6% (3)	100% (50)
$7x \simeq 7x$		2% (1)	6% (3)	–	–	100% (50)
$8x \simeq 8x$		–	–	–	–	90% (45)
$9x \simeq 9x$		–	–	–	–	88% (44)
$10x \simeq 10x$		–	–	–	–	68% (34)
$3x \leq_{\mathbb{N}} 3x$	$\mathbb{N}$	100% (2)	100% (2)	100% (2)	100% (2)	100% (2)
$4x \leq_{\mathbb{N}} 4x$		–	15% (3)	100% (20)	20% (4)	100% (20)
$5x \leq_{\mathbb{N}} 5x$		–	4% (2)	100% (50)	12% (6)	100% (50)
$1x \leq_{\mathbb{N}} 2x$		100% (1)	100% (1)	–	–	–
$2x \leq_{\mathbb{N}} 3x$		50% (1)	50% (1)	–	100% (2)	–
$3x \leq_{\mathbb{N}} 4x$		–	30% (3)	–	40% (4)	–
$4x \leq_{\mathbb{N}} 5x$		–	8% (4)	–	16% (8)	–
$5x \leq_{\mathbb{N}} 6x$		–	6% (3)	–	10% (5)	–
$1x \leq_{\mathbb{N}} 3x$		100% (2)	100% (2)	–	100% (2)	100% (2)
$2x \leq_{\mathbb{N}} 4x$		–	40% (2)	–	40% (2)	100% (5)
$3x \leq_{\mathbb{N}} 5x$		–	14% (4)	–	28% (8)	100% (28)
$4x \leq_{\mathbb{N}} 6x$		–	10% (5)	–	18% (9)	100% (50)
$5x \leq_{\mathbb{N}} 7x$		–	4% (2)	–	18% (9)	100% (50)
$1x \leq_{\mathbb{N}} 4x$		100% (5)	100% (5)	–	80% (4)	100% (5)
$2x \leq_{\mathbb{N}} 5x$		–	35% (5)	–	42% (6)	100% (14)
$3x \leq_{\mathbb{N}} 6x$		–	18% (9)	–	38% (19)	100% (50)
$4x \leq_{\mathbb{N}} 7x$		–	6% (3)	–	16% (8)	100% (50)
$5x \leq_{\mathbb{N}} 8x$		–	–	–	6% (3)	100% (50)
$1x \leq_{\mathbb{N}} 5x$		100% (14)	100% (14)	–	85% (12)	100% (14)
$2x \leq_{\mathbb{N}} 6x$		–	33% (14)	–	26% (11)	100% (42)
$3x \leq_{\mathbb{N}} 7x$		–	14% (7)	–	32% (16)	100% (50)
$4x \leq_{\mathbb{N}} 8x$		–	4% (2)	–	18% (9)	100% (50)
$5x \leq_{\mathbb{N}} 9x$		–	–	–	14% (7)	100% (50)
$\mathsf{pref}(3x, 3x)$	$\mathbb{L}$	100% (2)	50% (1)	–	–	100% (2)
$\mathsf{pref}(4x, 4x)$		–	25% (5)	–	–	100% (20)
$\mathsf{pref}(5x, 5x)$		–	2% (1)	–	4% (2)	100% (50)
$\mathsf{pref}(1x, 2x)$		100% (1)	100% (1)	–	–	–
$\mathsf{pref}(2x, 3x)$		–	50% (1)	–	50% (1)	–
$\mathsf{pref}(3x, 4x)$		–	20% (2)	–	20% (2)	–
$\mathsf{pref}(4x, 5x)$		–	8% (4)	–	8% (4)	–
$\mathsf{pref}(5x, 6x)$		–	–	–	–	–
$\mathsf{pref}(1x, 3x)$		100% (2)	100% (2)	–	50% (1)	100% (2)
$\mathsf{pref}(2x, 4x)$		20% (1)	40% (2)	–	20% (1)	100% (5)
$\mathsf{pref}(3x, 5x)$		–	14% (4)	–	14% (4)	100% (28)
$\mathsf{pref}(4x, 6x)$		–	6% (3)	–	8% (4)	100% (50)
$\mathsf{pref}(5x, 7x)$		–	2% (1)	–	2% (1)	100% (50)
$\mathsf{pref}(1x, 4x)$		100% (5)	100% (5)	–	40% (2)	100% (5)
$\mathsf{pref}(2x, 5x)$		–	35% (5)	–	21% (3)	100% (14)
$\mathsf{pref}(3x, 6x)$		–	14% (7)	–	12% (6)	100% (50)
$\mathsf{pref}(4x, 7x)$		–	4% (2)	–	4% (2)	100% (50)
$\mathsf{pref}(5x, 8x)$		–	–	–	4% (2)	100% (50)
$\mathsf{pref}(1x, 5x)$		100% (14)	100% (14)	–	42% (6)	100% (14)
$\mathsf{pref}(2x, 6x)$		–	33% (14)	–	21% (9)	100% (42)
$\mathsf{pref}(3x, 7x)$		–	16% (8)	–	16% (8)	100% (50)
$\mathsf{pref}(4x, 8x)$		–	10% (5)	–	12% (6)	100% (50)
$\mathsf{pref}(5x, 9x)$		–	–	–	–	100% (50)

Table 6.3: Experiments on 2,007 arithmetical problems.

corresponding clause $\neg L[t] \vee C$ was already selected as an induction candidate before, or (ii) if $\neg L[t] \vee C$ is selected as an induction candidate and the corresponding comparison literal was already selected before. To implement these rules efficiently, we should be able to efficiently retrieve comparison literals and literals selected for induction. To do so, we extended the indexing mechanism of Vampire to index such literals. We do not apply induction when the induction target formula $L[x]$ is a comparison having x as a top-level argument, for example, $x \leq_{\mathbb{Z}} t$, and allow to apply it to all other induction target formulas deemed to be suitable by other user-specified options.

Our (interval) downward/upward induction rules in Vampire are enabled by the new option `--induction int`. The options `--int_induction_interval infinite` and `--int_induction_interval finite` limit the enabled rules to downward/upward only, and interval downward/upward only, respectively. Further, `--int_induction_default_bound on` enables the more general rule which does not require bounds to be in the search space. Our new induction rules can also be controlled by other Vampire options for well-founded/structural induction, such as `--induction_on_complex_terms on`, which enables applying induction on any ground complex term. To improve Vampire's performance for integer induction, we combined our new induction rules with `--induction_on_complex_terms on` and also other options not specific to induction. We extended Vampire with a new mode scheduling various option configurations for integer induction, switched on by the option `--mode portfolio --schedule integer_induction`. Additionally, we introduced the option `--schedule induction` which uses either the integer induction configurations as for `--schedule integer_induction`, or structural induction configurations, or both, depending on the data types used in the problem/property to be proved.

6.2.2 Experimental Setup

We used two sets of examples: (i) benchmark sets LIA and UFLIA from the SMT-LIB collection [BFT16], consisting of, respectively, 607 and 10,137 examples, and (ii) 120 integer benchmarks from our new inductive benchmark set (see Subsection 5.2.2), similar to our motivating examples from Section 4.1.

We ran our experiments on computers with 32 cores (AMD Epyc 7502, 2.5 GHz) and 1 TB RAM. In all experiments we used the memory limit of 16 GB per problem. For the new benchmarks we used a 300-second time limit. For the experiments on the larger LIA and UFLIA sets we used a 10-second time limit.

In this section:

- Vampire$_o$ refers to the default version of Vampire.
- By Vampire$_i$ we denote our new version of Vampire, using integer induction rules (`--induction int`).

Problem set	Total count	CVC4	Z3	Vampire$_o$	Vampire$_i$	new compared to Vampire$_o$	new compared to Vampire$_o$, CVC4 and Z3
LIA	607	553	435	216	214	10	1
UFLIA	10137	7002	6705	6116	5796	99	44

Table 6.4: Comparison of solvers on SMT-LIB benchmarks.

- Vampire$_{ip}$ is like Vampire$_i$, but using additional options to run in the portfolio mode: scheduling various option configurations for integer induction (`--mode portfolio --schedule induction`).

For *experiments with the new benchmarks*, we note that Vampire$_o$ without integer induction cannot solve any of the problems. In this set of experiments, we therefore compared Vampire$_{ip}$ to the provers CVC4 [RK15] and ACL2 [KMM00], which are, to the best of our knowledge, the only two automated solvers supporting inductive reasoning with integers in addition to reasoning with theories and quantifiers. For CVC4, we used the *ig* configuration from [RK15]: `--quant-ind --quant-cf --conjecture-gen --conjecture-gen-per-round=3 --full-saturate-quant`. For ACL2, we used its default configuration. In the *experiments with the LIA and UFLIA benchmark sets of SMT-LIB*, we also used Z3 [DMB08] in the default configuration.

We ran CVC4, Z3, and all versions of Vampire on problems encoded in the SMT-LIB syntax [BFT16]. For running ACL2 on the new benchmarks, we translated problems into the functional program encoding syntax of ACL2.

6.2.3 Experimental Results

SMT-LIB benchmarks. First, we evaluated the improvements of integer induction in Vampire$_i$ when compared to Vampire$_o$, CVC4 and Z3 on the LIA and UFLIA sets of SMT-LIB [BFT16]. We aimed to verify that Vampire$_i$'s performance does not deteriorate due to adding integer induction, check whether Vampire$_i$ can solve problems that could not be solved automatically before, and to identify the best values for options related to integer induction. To this end, we picked five different strategies (e.g. using different saturation algorithms and selection functions) and used different combinations of induction options. Table 6.4 summarizes our results, showcasing that integer induction enabled Vampire$_i$ to solve over 100 new problems that Vampire$_o$ could not solve before (second to the last column of Table 6.4). Moreover, 45 of these problems were also new compared to CVC4 and Z3 (last column of Table 6.4), which most likely means that no theorem prover was able to prove them before.

In problems solved using integer induction, the integer induction rules were applied often: at least one of the interval induction rules was used in nearly 99% of problems, while one of the induction rules with one bound was used in nearly all problems. The interval

Problem set	Problem subset	Count	ACL2	CVC4	Vampire$_{ip}$
sum	*x_all*	1	0	0	1
	y_all	1	0	0	1
	x_leq	5	0	0	4
	y_geq	5	0	5	5
	subset total	12	0	5	11
val	*declared_mixed_ax-fin_conj-fin*	6	0	1	4
	declared_unint_ax-fin_conj-fin	3	0	0	3
	declared_inter_ax-all_conj-all	5	0	0	3
	declared_inter_ax-all_conj-geq	9	0	9	9
	declared_inter_ax-all_conj-leq	9	0	0	9
	declared_inter_ax-geq_conj-geq	13	0	13	10
	declared_inter_ax-leq_conj-leq	13	0	0	11
	*declared_unint_ax-all_**	7	0	0	7
	declared_unint_ax-geq_conj-geq	2	0	0	2
	declared_unint_ax-leq_conj-leq	2	0	0	2
	defined_inter_ax-all_conj-all	3	1	0	3
	defined_inter_ax-geq_conj-geq	3	2	3	3
	defined_inter_ax-leq_conj-leq	3	2	0	3
	*defined_unint_**	6	0	0	6
	subset total	84	5	26	75
power	*0_all*	4	0	0	4
	0_pos	4	0	0	4
	0_neg	4	0	0	4
	1_all	4	0	0	2
	1_pos	4	0	0	4
	1_neg	4	0	0	2
	subset total	24	0	0	20
all sets	combined total	120	5	31	106
all sets	uniquely solved	-	0	3	75

Table 6.5: Experiments with our new benchmarks from Table 5.1.

induction and induction rules were used on average 4559 and 1191 times, respectively. 89% of the proofs employed interval induction (67% upward, 29% downward), while 27% of the proofs used induction with one bound (22% upward, 8% downward). Additionally, over 64% of proofs only required one application of any induction rule.

Experiments with benchmarks from Subsection 5.2.2. Comparison results for Vampire$_{ip}$, ACL2 and CVC4 on our new benchmarks are displayed in Table 6.5, aggregated by benchmark subsets, as described in Table 5.1. We do not show Vampire$_o$ in the table, since without integer induction it cannot solve any of the problems.

The results show that in some cases ACL2 can perform upward and downward induction on integers, but only when using interpreted constants as a base case (that is, it cannot handle symbolic bounds). However, it can only do so if it also proves termination of the recursively defined function. It also has issues with reasoning about multiplication.

CVC4 has limited support for integer induction: it can apply upward induction but only when the base case is an interpreted constant. Since some problems seem to require induction with symbolic bounds, CVC4 is mostly able to either solve all problems in a subset, or none of them. The only exception is the subset *declared_mixed_ax-fin_conj-fin*, in which CVC4 solves one problem, which can be solved using upward induction with an interpreted constant as the base case.

Vampire$_{ip}$ does not have any conceptual problems with solving the benchmarks. However, since it uses axioms and inference rules rather than dedicated decision procedures for handling integers, it sometimes has issues with solving problems with large integer values. For example, for the infinite interval subset of the *val* benchmark set, the only problems Vampire$_{ip}$ did not solve were those containing the interpreted constant 100 or -100. Similarly, in the *power* benchmark set, the unsolved problems contained large numbers. Finally, in the *declared_mixed_ax-fin_conj-fin* subset, the two problems Vampire$_{ip}$ did not solve also required more sophisticated arithmetic reasoning. However, the inability to efficiently deal with large numbers is not an intrinsic problem of superposition theorem provers. Reasoning with quantifiers and theories is still in its infancy and major improvements are underway. For example, there are recent parallel developments in superposition and linear arithmetic [RSV21] that should improve this kind of reasoning in Vampire.

CHAPTER 7

Synthesis of Recursion-Free Programs

The contributions of this chapter are based on:

Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov. Program Synthesis in Saturation. In Brigitte Pientka and Cesare Tinelli, editors, Proc. of CADE, *volume 14132 of* LNCS, *pages 307–324, Cham, 2023. Springer [HKNV23],*

Petra Hozzová. Integrating Answer Literals with AVATAR for Program Synthesis. In Laura Kovács and Michael Rawson, editors, Proc. of the 7th and 8th Vampire Workshop, *volume 99 of* EPiC Series in Computing, *pages 13–20. EasyChair, 2024 [Hoz24]*

In this chapter we present a framework for synthesis of recursion-free programs using a first-order saturation-based theorem prover as a reasoning backend. We extract code from correctness proofs of functional specifications given as first-order formulas $\forall\overline{x}.\exists y.F[\overline{x}, y]$. These formulas state that "for all (program) inputs $\overline{x}$ there exists an output y such that the input-output relation (program computation) $F[\overline{x}, y]$ is valid". Given such a specification, we synthesize a recursion-free program while also deriving a proof certifying that the program satisfies the specification.

The programs we synthesize are built using first-order theory terms extended with `if − then − else` constructors. To ensure that our programs yield computational models, i.e., that they can be evaluated for given values of input variables $\overline{x}$, we restrict the programs we synthesize to only contain *computable* symbols.

Briefly, in order to synthesize a recursion-free program, we prove its functional specification using saturation-based theorem proving [NR01, KV13]. We extend saturation-based proof search with answer literals [Gre69], allowing us to track substitutions into the output variable y of the specification. These substitutions correspond to the sought program fragments and are conditioned on clauses they are associated with in the proof. When we derive a clause corresponding to a program branch `if` C `then` r, where C is a condition and r a term and both C, r are computable, we store it and continue proof search assuming that $\neg C$ holds; we refer to such conditions C as (program) branch conditions. The saturation process for both proof search and code construction terminates when the conjunction of negations of the collected branch conditions becomes unsatisfiable. Then we synthesize the final program satisfying the given (and proved) specification by assembling the recorded program branches (see e.g. Examples 7.1, 7.9, and 7.10).

The main challenges of making our approach effective come with (i) integrating the construction of the programs with `if`$-$`then`$-$`else` into the proof search, turning thus proof search into *program search/synthesis*, and (ii) guiding program synthesis to derive only computable branch conditions and programs.

We start this chapter with a terminology overview in Section 7.1, and then we present an illustrative example in Section 7.2. In Section 7.3 we formalize the semantics for clauses with answer literals and introduce a *saturation-based algorithm for program synthesis* based on this semantics. We prove that, given a sound inference system, our saturation algorithm derives correct and computable programs. Next, in Section 7.4, we define the properties of a sound inference calculus in order to make the calculus suitable for our saturation-based algorithm for program synthesis. We accordingly extend the superposition calculus and define a class of substitutions to be used within the extended calculus; we refer to these substitutions as *computable unifiers*. Then in Section 7.5 we extend a first-order unification algorithm to find computable unifiers to be further used in saturation-based program synthesis. In Section 7.6 we integrate our saturation-based algorithm with with the Avatar framework [Vor14], enabling efficient splitting and theory reasoning.

Later in Chapter 9 we describe the implementation of our work in the Vampire prover [KV13] and evaluate our synthesis approach on a number of examples, complementing other techniques in the area (Section 9.1). For example, our results demonstrate the applicability of our work on synthesizing programs for specifications that cannot be even encoded in the SyGuS syntax [PPR$^+$21].

7.1 Computable Symbols and Programs

We distinguish between *computable* and *uncomputable* symbols in the signature. The set of computable symbols is given as part of the specification. Intuitively, a symbol is computable if it can be evaluated and hence is allowed to occur in a synthesized program. A term or a literal is *computable* if all symbols it contains are computable. A symbol, term, or literal is *uncomputable* if it is not computable.

A *functional specification*, or simply just a *specification*, is a formula

$$\forall \overline{x}. \exists y. F[\overline{x}, y]. \tag{7.1}$$

$$\forall x.\, \mathsf{inv}(x)*x \simeq \mathsf{e} \;\text{(G1)} \qquad \forall x.\, \mathsf{e}*x \simeq x \;\text{(G2)} \qquad \forall x,y,z.\, x*(y*z) \simeq (x*y)*z \;\text{(G3)}$$

Figure 7.1: Axioms defining a group. Uninterpreted function symbols $\mathsf{inv}(\cdot), \mathsf{e}, *$ represent the inverse, the identity element, and the group operation, respectively.

The variables $\overline{x}$ of a specification (7.1) are called *input variables*. Note that while we use specifications with a single variable y, our work can analogously be used with a tuple of variables $\overline{y}$ in (7.1).

Let $\overline{\sigma}$ denote a tuple of skolem constants. Consider a computable term $r[\overline{\sigma}]$ such that the instance $F[\overline{\sigma}, r[\overline{\sigma}]]$ of (7.1) holds. Since $\overline{\sigma}$ are fresh skolem constants, the formula $\forall \overline{x}.F[\overline{x}, r[\overline{x}]]$ also holds; we call such $r[\overline{x}]$ a *program* for (7.1) and say that the program $r[\overline{x}]$ *computes a witness* of (7.1).

Further, if $\forall \overline{x}.(F_1 \wedge \ldots \wedge F_n \rightarrow F[\overline{x}, r[\overline{x}]])$ holds for computable formulas $F_1, \ldots, F_n$, we write $\langle r[\overline{x}], \bigwedge_{i=1}^{n} F_i \rangle$ to refer to a *program with conditions* $F_1, \ldots, F_n$ for (7.1). In the sequel, we refer to (parts of) programs with conditions also as *conditional branches*. In Section 7.3 we show how to build programs for (7.1) by composing programs with conditions for (7.1) (see Corollary 7.4).

7.2 Illustrative Example

Let us illustrate our approach to program synthesis. We use answer literals (see Section 2.4) in saturation to construct programs with conditions while proving specifications of the form (7.1). By adding an answer literal to the skolemized negation of (7.1), we obtain

$$\forall y.(\neg F[\overline{\sigma}, y] \vee \mathsf{ans}(y)), \tag{7.2}$$

where $\overline{\sigma}$ are the skolemized input variables $\overline{x}$. When we derive a unit clause $\mathsf{ans}(r[\overline{\sigma}])$ during saturation, where $r[\overline{\sigma}]$ is a computable term, we construct a program for (7.1) from the definite answer $r[\overline{\sigma}]$ by replacing $\overline{\sigma}$ with the input variables $\overline{x}$, obtaining the program $r[\overline{x}]$. Hence, deriving computable definite answers by saturation allows us to synthesize programs for specifications.

Example 7.1. Consider the group theory axioms (G1)–(G3) of Figure 7.1. We are interested in synthesizing a program for the following specification:

$$\forall x.\exists y.\; x * y \simeq \mathsf{e} \tag{7.3}$$

In this example we assume that all symbols are computable. To synthesize a program for (7.3), we preprocess the specification by adding an answer literal to the skolemized negation of (7.3) and convert the resulting formula to CNF. We consider the set S of clauses containing the obtained CNF and the axioms (G1)–(G3). We saturate S using $\mathbb{S}$up and obtain the following derivation:

1. $\sigma * y \not\simeq \mathsf{e} \vee \mathsf{ans}(y)$ [preprocessed specification]
2. $\mathsf{inv}(x) * (x * y) \simeq \mathsf{e} * y$ [Sup (G1), (G3)]
3. $\mathsf{inv}(x) * (x * y) \simeq y$ [Sup (G2), 2]
4. $\mathsf{inv}(\mathsf{inv}(x)) * y \simeq x * y$ [Sup 3, 3[1]]
5. $\mathsf{e} \simeq x * \mathsf{inv}(x)$ [Sup 4, (G1)]
6. $\mathsf{ans}(\mathsf{inv}(\sigma))$ [BR 5, 1]

Using the above derivation, we construct a program for the functional specification (7.3) as follows: we replace σ in the definite answer $\mathsf{inv}(\sigma)$ by x, yielding the program $\mathsf{inv}(x)$. Note that for each input x, our synthesized program computes the inverse $\mathsf{inv}(x)$ of x as an output. In other words, our synthesized program for (7.3) ensures that each group element x has a right inverse $\mathsf{inv}(x)$. □

While Example 7.1 yields a definite answer within saturation-based proof search, our work supports the synthesis of more complex recursion-free programs (see Examples 7.9 and 7.10) by composing program fragments derived in the program search (Section 7.3) as well as by using answer literals with `if−then−else` to effectively handle disjunctive answers (Section 7.4).

7.3 Program Synthesis with Answer Literals

We now introduce our approach to saturation-based program synthesis using answer literals (Algorithm 7.1). Here we focus on recursion-free program synthesis, which we in Chapter 8 extend to recursive synthesis, and present our work in a more general setting. Namely, we consider functional specifications whose validity may depend on additional assumptions (e.g. additional program requirements) $A_1, \ldots, A_n$, where each A_i is a closed formula:

$$A_1 \wedge \ldots \wedge A_n \rightarrow \forall \overline{x}. \exists y. F[\overline{x}, y] \tag{7.4}$$

Note that specification (7.1) is a special case of (7.4). However, since $A_1, \ldots, A_n$ are closed formulas, (7.4) is equivalent to $\forall \overline{x}. \exists y. (A_1 \wedge \ldots \wedge A_n \rightarrow F[\overline{x}, y])$, which is a special case of (7.1).

Given a functional specification (7.4), we use answer literals to synthesize programs with conditions (Section 7.3.1) and extend saturation-based proof search to reason about

[1]For clarity, consider two instances of 3 with differently named variables:

$$3.\ \mathsf{inv}(x) * (x * y) \simeq y$$
$$3'.\ \mathsf{inv}(z) * (z * w) \simeq w$$

We rewrite the term $z * w$ in $3'$ using the equality 3 and the substitution $\{z \mapsto \mathsf{inv}(x), w \mapsto x * y\}$. The instance of $3'$ we rewrite is $\mathsf{inv}(\mathsf{inv}(x)) * (\mathsf{inv}(x) * (x * y)) \simeq x * y$. After rewriting we obtain 4.

answer literals (Section 7.3.2). For doing so, we add the answer literal $\mathsf{ans}(y)$ to the skolemized negation of (7.4) and obtain

$$A_1 \wedge \ldots \wedge A_n \wedge \forall y.(\neg F[\overline{\sigma}, y] \vee \mathsf{ans}(y)). \tag{7.5}$$

We saturate the CNF of (7.5) while ensuring that answer literals are not selected within the inference rules used in saturation. We guide saturation-based proof search to derive clauses $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$, where $C[\overline{\sigma}]$ and $r[\overline{\sigma}]$ are computable.

7.3.1 From Answer Literals to Programs

Our next result ensures that, if we derive the clause $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$, the term $r[\overline{\sigma}]$ is a definite answer under the assumption $\neg C[\overline{\sigma}]$ (Theorem 7.2). We note that we do not terminate saturation-based program synthesis once a clause $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ is derived. We rather record the program $r[\overline{x}]$ with condition $\neg C[\overline{x}]$ (and possibly also other conditions), replace clause $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ by $C[\overline{\sigma}]$, and continue saturation (Corollary 7.3). As a result, upon establishing validity of (7.4), we synthesized a program for (7.4) (Corollary 7.4).

Theorem 7.2 (Semantics of Clauses with Answer Literals)**.** Let C be a clause not containing an answer literal. Assume that, using a saturation algorithm based on a sound inference system $\mathcal{I}$, the clause $C \vee \mathsf{ans}(r[\overline{\sigma}])$ is derived from the set of clauses consisting of initial assumptions $A_1, \ldots, A_n$, the clausified formula $\mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee \mathsf{ans}(y))$ and additional assumptions $C_1, \ldots, C_m$. Then,

$$A_1, \ldots, A_n, C_1, \ldots, C_m \vdash C, F[\overline{\sigma}, r[\overline{\sigma}]].$$

That is, under the assumptions $C_1, \ldots, C_m, \neg C$, the computable term $r[\overline{\sigma}]$ provides a definite answer to (7.4).

Proof. We consider the calculus that was used for deriving $C \vee \mathsf{ans}(r[\overline{\sigma}])$, but with lifted ordering and selection conditions. I.e., we allow application of the rules regardless of the term ordering and of which literals are selected. Since the soundness of the calculus does not depend on these side conditions, the calculus without the conditions is sound as well. Now, since ans is uninterpreted, we can replace $\mathsf{ans}(y)$ by $y \not\simeq r[\overline{\sigma}]$, and obtain a derivation of $C \vee r[\overline{\sigma}] \not\simeq r[\overline{\sigma}]$ from $A_1, \ldots, A_n, \forall y.\mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r[\overline{\sigma}])$ using the calculus without the conditions.[2]

We want to show that

$$\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{m} C_i \rightarrow C \vee F[\overline{\sigma}, r[\overline{\sigma}]] \tag{7.6}$$

is valid. Hence, we need to show that in each interpretation, in which the antecedent is true, also the consequent is true. Let us consider such an interpretation I. We

[2]The derivation might not have been possible in the calculus with the ordering and selection conditions due to replacing the positive literal $\mathsf{ans}(y)$ with the negative literal $y \not\simeq r[\overline{\sigma}]$ containing different symbols.

distinguish two cases. First, assume that $\forall y.\mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r[\overline{\sigma}])$ is true in I. Then since all assumptions from which we derived $C \vee r[\overline{\sigma}] \not\simeq r[\overline{\sigma}]$ are true in I and since the inference system is sound, also $C \vee r[\overline{\sigma}] \not\simeq r[\overline{\sigma}]$ is true. That clause is equivalent to C, hence C is true, which makes the consequent of (7.6) true. Second, assume that $\forall y.\mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r[\overline{\sigma}])$ is false in I. Then its negation, $\neg\forall y.\mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r[\overline{\sigma}])$, equivalent to $\exists y.(F[\overline{\sigma}, y] \wedge y \simeq r[\overline{\sigma}])$, equivalent to $F[\overline{\sigma}, r[\overline{\sigma}]]$ must be true in I. Hence, the consequent of (7.6) is true also in this case. Therefore (7.6) is valid. □

We further use Theorem 7.2 to synthesize programs with conditions for (7.4).

Corollary 7.3 (Programs with Conditions)**.** Let $r[\overline{\sigma}]$ be a computable term and $C[\overline{\sigma}]$ a ground computable clause not containing an answer literal. Assume that clause $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ is derived from the set of initial clauses $A_1, \dots, A_n$, the clausified formula $\mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee \mathsf{ans}(y))$ and additional ground computable assumptions $C_1[\overline{\sigma}], \dots, C_m[\overline{\sigma}]$, by using saturation based on a sound inference system $\mathcal{I}$. Then,

$$\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}] \rangle$$

is a program with conditions for (7.4).

Proof. From Theorem 7.2 follows that $\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{m} C_i[\overline{\sigma}] \rightarrow C[\overline{\sigma}] \vee F[\overline{\sigma}, r[\overline{\sigma}]]$ holds. Since $\overline{\sigma}$ are fresh uninterpreted constants, we obtain that $\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{m} C_i[\overline{x}] \rightarrow C[\overline{x}] \vee F[\overline{x}, r[\overline{x}]]$ is valid as well, and that is equivalent to $\bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}] \rightarrow (\bigwedge_{i=1}^{n} A_i \rightarrow F[\overline{x}, r[\overline{x}]])$. Therefore $\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}] \rangle$ is a program with conditions for $A_1 \wedge \ldots \wedge A_n \rightarrow \forall \overline{x}.\exists y.F[\overline{x}, y]$. □

Note that a program with conditions $\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}] \rangle$ corresponds to a conditional (program) branch `if` $\bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}]$ `then` $r[\overline{x}]$: only if the condition $\bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}]$ is valid, then $r[\overline{x}]$ is computed for (7.4).

We use programs with conditions $\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}] \rangle$ to finally synthesize a program for (7.4). To this end, we use Corollary 7.3 to derive programs with conditions, and once their conditions cover all possible cases given the initial assumptions $A_1, \dots, A_n$, we compose them into a program for (7.4).

Corollary 7.4 (From Programs with Conditions to Programs for (7.4))**.** Let $P_1[\overline{x}], \dots, P_k[\overline{x}]$, where $P_i[\overline{x}] = \langle r_i[\overline{x}], \bigwedge_{j=1}^{i-1} C_j[\overline{x}] \wedge \neg C_i[\overline{x}] \rangle$, be programs with conditions for (7.4), such that $\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{k} C_i[\overline{x}]$ is unsatisfiable. Then $P[\overline{x}]$, given by

$$\begin{aligned} P[\overline{x}] := {} & \texttt{if } \neg C_1[\overline{x}] \texttt{ then } r_1[\overline{x}] \\ & \texttt{else if } \neg C_2[\overline{x}] \texttt{ then } r_2[\overline{x}] \\ & \quad \dots \\ & \texttt{else if } \neg C_{k-1}[\overline{x}] \texttt{ then } r_{k-1}[\overline{x}] \\ & \texttt{else } r_k[\overline{x}], \end{aligned} \tag{7.7}$$

is a program for (7.4).

Proof. For any interpretation I and any variable assignment v, let p be the smallest index such that $\neg C_p[\overline{x}]$ holds in I under v, but all $\neg C_j[\overline{x}]$, where $1 \leq j < p$, do not hold in I under v. Since $\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{k} C_i[\overline{x}]$ is unsatisfiable, under the assumptions $A_1, \ldots, A_n$ such a p has to exist. Then in I under v and under the assumptions $A_1, \ldots, A_n$, the interpretation of $P[\overline{x}]$ is the same as the interpretation of $r_p[\overline{x}]$.

Further, since $\bigwedge_{j=1}^{p-1} C_j[\overline{x}] \wedge \neg C_p[\overline{x}]$ is the condition for $P_p[\overline{x}]$, from the definition of a program with conditions we obtain that $A_1 \wedge \cdots \wedge A_n \rightarrow F[\overline{x}, r_p[\overline{x}]]$ holds in I under v. Hence also $A_1 \wedge \cdots \wedge A_n \rightarrow F[\overline{x}, P[\overline{x}]]$ holds in I under v.

Finally, since this argument holds for any I and v, and since all $A_1, \ldots, A_n$ are closed formulas, also $A_1 \wedge \cdots \wedge A_n \rightarrow \forall \overline{x}. F[\overline{x}, P[\overline{x}]]$ holds. Therefore $P[\overline{x}]$ is a program for (7.4). □

Note that since the conditional branches of (7.7) cover all possible cases to be considered over $\overline{x}$, we do not need the condition `if` $\neg C_k$. In particular, if $k = 1$, i.e. $\bigwedge_{i=1}^{n} A_i \wedge C_1[\overline{x}]$ is unsatisfiable, then the synthesized program for (7.4) is $r_1[\overline{x}]$.

7.3.2 Saturation-Based Program Synthesis

Our program synthesis results from Theorem 7.2, Corollary 7.3 and Corollary 7.4 rely upon a saturation algorithm using a sound (but not necessarily complete) inference system $\mathcal{I}$. In this section, we present our modifications to extend state-of-the-art saturation algorithms with answer literal reasoning, allowing us to derive clauses $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$, where both $C[\overline{\sigma}]$ and $r[\overline{\sigma}]$ are computable. In Sections 7.4–7.5 we then describe modifications of the inference system $\mathcal{I}$ to implement rules over clauses with answer literals.

Our saturation algorithm is given in Algorithm 7.1. In a nutshell, we use Corollary 7.3 to construct programs from clauses $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ and replace clauses $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ by $C[\overline{\sigma}]$ (lines 7–10 of Algorithm 7.1). The newly added computable assumptions $C[\overline{\sigma}]$ are used to guide saturation towards deriving programs with conditions where the conditions contain $C[\overline{x}]$; these programs with conditions are used for synthesizing programs for (7.4), as given in Corollary 7.4.

Compared to a standard saturation algorithm used in first-order theorem proving (e.g. lines 4–5 of Algorithm 7.1), Algorithm 7.1 implements additional steps for processing newly derived clauses $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ with answer literals (lines 6-10). As a result, Algorithm 7.1 establishes not only the validity of the specification (7.4) but also synthesizes a program (lines 12-13). Throughout the algorithm, we maintain a set $\mathcal{P}$ of programs with conditions derived so far and a set $\mathcal{C}$ of additional assumptions. For each new clause C_i, we check if it is in the form $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ where $C[\overline{\sigma}]$ is ground and computable (line 7). If yes, we construct a program with conditions $\langle r[\overline{x}], \bigwedge_{C' \in \mathcal{C}} C' \wedge \neg C[\overline{x}] \rangle$, extend $\mathcal{C}$ with the additional assumption $C[\overline{x}]$, and replace C_i by $C[\overline{\sigma}]$ (lines 8-10). Then, when we

Algorithm 7.1: Saturation Loop for Program Synthesis

```
initial set of clauses $S := \{\mathsf{cnf}(A_1 \wedge \ldots \wedge A_n \wedge \forall y.(\neg F[\overline{\sigma}, y] \vee \mathsf{ans}(y)))\}$
initial sets of additional assumptions $\mathcal{C} := \emptyset$ and programs $\mathcal{P} := \emptyset$
repeat
  Select clause $G \in S$
  Derive consequences $C_1, \ldots, C_n$ of $G$ and formulas from $S$ using rules of $\mathcal{I}$
  for each $C_i$ do
    if $C_i = (C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}]))$ and $C[\overline{\sigma}]$ is ground and computable then
      $\mathcal{P} := \mathcal{P} \cup \{\langle r[\overline{x}], \bigwedge_{C' \in \mathcal{C}} C' \wedge \neg C[\overline{x}] \rangle\}$        /* Corollary 7.3 */
      $\mathcal{C} := \mathcal{C} \cup \{C[\overline{x}]\}$
      $C_i := C[\overline{\sigma}]$
  $S := S \cup \{C_1, \ldots, C_n\}$
  if $\square \in S$ then
    return program (7.7) for specification (7.4),
           derived from $\mathcal{P}$        /* Corollary 7.4 */
```

derive the empty clause, we construct the final program as follows. We first collect all clauses that participated in the derivation of $\square$. We use this clause collection to filter the programs in $\mathcal{P}$ – we only keep a program originating from a clause $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ if the condition $C[\overline{\sigma}]$ was used in the proof, obtaining programs $P_1, \ldots, P_k$. From $P_1, \ldots, P_k$ we then synthesize the final program P using the construction (7.7) from Corollary 7.4.

Remark. Compared to [Tam95] where potentially large programs (with conditions) are tracked in answer literals, Algorithm 7.1 removes answer literals from clauses and constructs the final program only after saturation found a refutation of the negated (7.4). Our approach has two advantages: first, we do not have to keep track of potentially many large terms using `if − then − else`, which might slow down saturation-based program synthesis. Second, our work can naturally be integrated with clause splitting techniques within saturation (see Section 7.6).

7.4 Superposition with Answer Literals

We note that our saturation-based program synthesis approach is not restricted to a specific calculus. Algorithm 7.1 can thus be used with *any sound* set of inference rules, including theory-specific inference rules, e.g. [KKR+23], as long as the rules allow derivation of clauses in the form $C \vee \mathsf{ans}(r)$, where C, r are computable and C is ground. I.e., the rules should only derive clauses with at most one answer literal, and should not introduce uncomputable symbols into answer literals.

In this section we present changes tailored to the superposition calculus $\mathbb{S}\mathrm{up}$, yet, without changing the underlying saturation process of Algorithm 7.1. We first introduce the notion of an abstract unifier [RSV18] and define a computable unifier – a mechanism for dealing with the uncomputable symbols in the reasoning instead of introducing them into the programs. The use of such a unifier in any sound calculus is explained, with particular focus on the $\mathbb{S}$up calculus.

Definition 7.5 (Abstract Unifier [RSV18])**.** An *abstract unifier* of two expressions E_1, E_2 is a pair (θ, D) such that:

1. θ is a substitution and D is a (possibly empty) disjunction of disequalities,
2. $(D \vee E_1 \simeq E_2)\theta$ is valid in the underlying theory.

$\square$

Intuitively speaking, an abstract unifier combines disequality constraints D with a substitution θ such that the substitution is a unifier of E_1, E_2 if the constraints D are not satisfied.

Definition 7.6 (Computable Unifier)**.** A *computable unifier* of two expressions E_1, E_2 with respect to an expression E_3 is an abstract unifier (θ, D) of E_1, E_2 such that the expression $E_3\theta$ is computable. $\square$

For example, let f be computable and g uncomputable. Then $(\{y \mapsto f(z)\}, z \not\simeq g(x))$ is a computable unifier of the terms $f(g(x)), y$ with respect to $f(y)$. Further, $(\{y \mapsto f(g(x))\}, \emptyset)$ is an abstract unifier of the same terms, but not a computable unifier with respect to $f(y)$.

Ensuring computability of answer literal arguments. We modify the rules of a sound inference system $\mathcal{I}$ to use computable unifiers with respect to the answer literal argument instead of unifiers. Since a computable unifier may entail disequality constraints D, we add D to the conclusions of the inference rules. That is, for an inference rule of $\mathcal{I}$ as below

$$\frac{C_1 \quad \cdots \quad C_n}{C\theta}\ , \tag{7.8}$$

where θ is a substitution such that $E\theta \simeq E'\theta$ holds for some expressions E, E', we extend $\mathcal{I}$ with the following n inference rules with computable unifiers:

$$\frac{C_1 \vee \mathsf{ans}(r) \quad C_2 \quad \cdots \quad C_n}{(\underline{D} \vee C \vee \mathsf{ans}(r))\theta'} \qquad \cdots \qquad \frac{C_1 \quad C_2 \quad \cdots \quad C_n \vee \mathsf{ans}(r)}{(\underline{D} \vee C \vee \mathsf{ans}(r))\theta'}\ , \tag{7.9}$$

where (θ', D) is a computable unifier of E, E' with respect to r and none of $C_1, \ldots, C_n$ contains an answer literal. We obtain the following result.

Lemma 7.7 (Soundness of Inferences with Answer Literals)**.** If the rule (7.8) is sound, the rules (7.9) are sound as well.

Proof. We will prove the soundness of the new rule

$$\frac{C_1 \vee \mathsf{ans}(r) \quad C_2 \quad \cdots \quad C_n}{(D \vee C \vee \mathsf{ans}(r))\theta'} \quad , \tag{7.10}$$

where (θ', D) is a computable unifier of E, E' with respect to r, and none of $C_1, \ldots, C_n$ contains an answer literal. The proof of soundness of the other new rules of (7.9) is analogous.

Assume interpretation I to be a model of the universal closures of the premises of (7.10), but not a model of the universal closure of its conclusion. Then $D\theta', C\theta'$ and $\mathsf{ans}(r)\theta'$ are false in I. From $D\theta'$ being false in I and from (θ', D) being an abstract unifier follows that $E\theta' \simeq E'\theta'$ holds. We can therefore set $\theta := \theta'$. From the soundness of (7.8) and $C\theta'$ being false in I then follows that some of $C_1, \ldots, C_n$ is false in I. However, none of $C_2, \ldots, C_n$ can be false in I, because we assumed all premises of (7.10) to be true in I. Hence, C_1 is false in I. Further, from $\mathsf{ans}(r)\theta'$ being false in I follows that $\mathsf{ans}(r)$ is false in I. However, that means that $C_1 \vee \mathsf{ans}(r)$ is false in I, which contradicts the assumption that the universal closures of all premises of rule (7.10) are true in I.

Hence, the rule (7.10) is sound. □

We note that we keep the original rule (7.8) in $\mathcal{I}$, but impose that none of its premises $C_1, \ldots, C_n$ contains an answer literal. Clearly, neither the such modified rule (7.8) nor the new rules (7.9) introduce uncomputable symbols into answer literals. Rather, these rules add disequality constraints D into their conclusions and immediately select D for further applications of inference rules. Such a selection guides the saturation process in Algorithm 7.1 to first discharge the constraints D containing uncomputable symbols with the aim of deriving a clause $C \vee \mathsf{ans}(r)$ where C is computable. If C is ground, the clause $C \vee \mathsf{ans}(r)$ is then converted into a program with conditions using Corollary 7.3.

Superposition with answer literals. We make the inference rule modifications (7.8), together with the addition of new rules (7.9), for each inference rule of the $\mathbb{S}$up calculus from Figure 2.2. Further, we also ensure that rules with multiple premises, when applied on several premises containing answer literals, *derive clauses with at most one answer literal.* We therefore introduce the following two rule modifications:

1. We use the `if − then − else` constructor to combine answer literals of premises, by adapting the use of `if–then–else` within binary resolution [LWC74, MW80, Tam95] to superposition rules.

2. We use an answer literal from only one of the rule premises in the rule conclusion and add new disequality constraint $r \not\simeq r'$ between the premises' answer literal arguments, similar to the constraints D of the computable unifier. Analogously to the computable unifier constraints, we immediately select this disequality constraint $r \not\simeq r'$.

Superposition (Sup):

$$\frac{\underline{s \simeq t} \lor C \lor \mathsf{ans}(r) \quad \underline{L[s']} \lor C' \lor \mathsf{ans}(r')}{(\underline{D} \lor L[t] \lor C \lor C' \lor \mathsf{ans}(\texttt{if } s \simeq t \texttt{ then } r' \texttt{ else } r))\theta} \qquad \frac{\underline{s \simeq t} \lor C \lor \mathsf{ans}(r) \quad \underline{L[s']} \lor C' \lor \mathsf{ans}(r')}{(\underline{D} \lor \underline{r \not\simeq r'} \lor L[t] \lor C \lor C' \lor \mathsf{ans}(r))\theta}$$

$$\frac{\underline{s \simeq t} \lor C \lor \mathsf{ans}(r) \quad \underline{u[s'] \not\simeq u'} \lor C' \lor \mathsf{ans}(r')}{(\underline{D} \lor u[t] \not\simeq u' \lor C \lor C' \lor \mathsf{ans}(\texttt{if } s \simeq t \texttt{ then } r' \texttt{ else } r))\theta} \qquad \frac{\underline{s \simeq t} \lor C \lor \mathsf{ans}(r) \quad \underline{u[s'] \simeq u'} \lor C' \lor \mathsf{ans}(r')}{(\underline{D} \lor \underline{r \not\simeq r'} \lor u[t] \simeq u' \lor C \lor C' \lor \mathsf{ans}(r))\theta}$$

$$\frac{\underline{s \simeq t} \lor C \lor \mathsf{ans}(r) \quad \underline{u[s'] \simeq u'} \lor C' \lor \mathsf{ans}(r')}{(\underline{D} \lor u[t] \simeq u' \lor C \lor C' \lor \mathsf{ans}(\texttt{if } s \simeq t \texttt{ then } r' \texttt{ else } r))\theta} \qquad \frac{\underline{s \simeq t} \lor C \lor \mathsf{ans}(r) \quad \underline{u[s'] \not\simeq u'} \lor C' \lor \mathsf{ans}(r')}{(\underline{D} \lor \underline{r \not\simeq r'} \lor u[t] \not\simeq u' \lor C \lor C' \lor \mathsf{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, s' w.r.t. the argument of the answer literal in the rule conclusion (i.e. `if` $s \simeq t$ `then` r' `else` r for the left-column rules, and r for the others); (rules on the first line only) $L[s']$ is not an equality literal; and (rules on the second and third line only) $u'\theta \not\succeq u[s']\theta$.

Binary resolution (BR):

$$\frac{\underline{A} \lor C \lor \mathsf{ans}(r) \quad \neg\underline{A'} \lor C' \lor \mathsf{ans}(r')}{(\underline{D} \lor C \lor C' \lor \mathsf{ans}(\texttt{if } A \texttt{ then } r' \texttt{ else } r))\theta} \qquad \frac{\underline{A} \lor C \lor \mathsf{ans}(r) \quad \neg\underline{A'} \lor C' \lor \mathsf{ans}(r')}{(\underline{D} \lor \underline{r \not\simeq r'} \lor C \lor C' \lor \mathsf{ans}(r))\theta}$$

where (θ, D) is a computable unifier of A, A' w.r.t. (first rule) `if` A `then` r' `else` r or (second rule) r.

Factoring (Fac):

$$\frac{\underline{A} \lor \underline{A'} \lor C \lor \mathsf{ans}(r)}{(\underline{D} \lor A \lor C \lor \mathsf{ans}(r))\theta}$$

where (θ, D) is a computable unifier of A, A' w.r.t. r.

Equality resolution (ER):

$$\frac{\underline{s \not\simeq t} \lor C \lor \mathsf{ans}(r)}{(\underline{D} \lor C \lor \mathsf{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, t w.r.t. r.

Equality factoring (EF):

$$\frac{\underline{s \simeq t} \lor \underline{s' \simeq t'} \lor C \lor \mathsf{ans}(r)}{(\underline{D} \lor s \simeq t \lor t \not\simeq t' \lor C \lor \mathsf{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, s' w.r.t. r; $t\theta \not\succeq s\theta$; and $t'\theta \not\succeq t\theta$.

Figure 7.2: Selected rules of the extended superposition calculus $\mathbb{S}$up for reasoning with answer literals, with underlined literals being selected.

The resulting extension of the $\mathbb{S}$up calculus with answer literals is given in Figure 7.2. In addition to the rules of Figure 7.2, the extended calculus contains rules constructed as (7.9) for superposition and binary resolution rules of Figure 2.2. Using Lemma 7.7, we conclude the following.

Lemma 7.8 (Soundness of $\mathbb{S}$up with Answer Literals)**.** The inference rules of the extended $\mathbb{S}$up calculus with answer literals (from Figure 7.2) are sound.

Proof. Soundness of the factoring, equality factoring, and equality resolution rules follows from Lemma 7.7.

We will prove soundness for the first superposition rule and the second binary resolution rule. The proofs for other superposition and binary resolution rules are analogous.

For clarity we recall the first superposition rule of Figure 7.2:

$$\frac{s \simeq t \lor C \lor \mathsf{ans}(r) \quad L[s'] \lor C' \lor \mathsf{ans}(r')}{(D \lor L[t] \lor C \lor C' \lor \mathsf{ans}(\texttt{if } s{\simeq}t \texttt{ then } r' \texttt{ else } r))\theta}$$

Assume interpretation I to be a model of the universal closures of the premises of the rule, but not a model of the universal closure of its conclusion. Then there is some variable assignment v such that $(D \lor L[t] \lor C \lor C' \lor \mathsf{ans}(\texttt{if } s{\simeq}t \texttt{ then } r' \texttt{ else } r))\theta$ is false in I under v. Let v' be a variable assignment that assigns to each variable x the value that $x\theta$ has in I under v. Then:

1. From $L[t]\theta, C\theta, C'\theta$ being false in I under v follows that $L[t], C, C'$ are false in I under v'.

2. Since $D\theta$ is false in I under v, from (θ, D) being an abstract unifier of s, s' follows that $s\theta \simeq s'\theta$ is true in I under v, and therefore s, s' have the same interpretation in I under v'. Then consider two cases:

   a) $s \simeq t$ is true in I under v' and $s\theta \simeq t\theta$ is true in I under v. Then from $\mathsf{ans}(\texttt{if } s{\simeq}t \texttt{ then } r' \texttt{ else } r)\theta$ being false in I under v follows that $\mathsf{ans}(r')\theta$ is false in I under v and therefore $\mathsf{ans}(r')$ is false in I under v'. Also from $s \simeq t$ being true in I under v', 1., and 2. follows that $L[s']$ is false in I under v'. Then the whole second premise of the rule is false in I under v', which is a contradiction with the assumption that I is a model of its universal closure.

   b) $s \simeq t$ is false in I under v' and $s\theta \simeq t\theta$ is false in I under v. This case leads similarly to the first premise being false, in contradiction with the assumption.

Therefore the first superposition rule is sound.

For clarity we recall the second binary resolution rule of Figure 7.2:

$$\frac{A \lor C \lor \mathsf{ans}(r) \quad \neg A' \lor C' \lor \mathsf{ans}(r')}{(D \lor r {\not\simeq} r' \lor C \lor C' \lor \mathsf{ans}(r))\theta}$$

Assume interpretation I to be a model of the universal closures of the premises of the rule, but not a model of the universal closure of its conclusion. Then there is some variable assignment v such that $(D \lor r {\not\simeq} r' \lor C \lor C' \lor \mathsf{ans}(r))\theta$ is false in I under v. Let v' be a variable assignment that assigns to each variable x the value that $x\theta$ has in I under v. Then:

1. From $r\theta \not\simeq r'\theta, C\theta, C'\theta$ being false in I under v follows that $r \not\simeq r', C, C'$ are false in I under v'. Therefore r, r' have the same interpretation in I under v'.

2. Since $\mathsf{ans}(r)\theta$ is false in I under v, also $\mathsf{ans}(r)$ is false in I under v'. Then from 1. follows that $\mathsf{ans}(r')$ is also false in I under v'.

3. Since $D\theta$ is false in I under v, from (θ, D) being an abstract unifier of A, A' follows that $A\theta, A'\theta$ have the same interpretation in I under v, and therefore A, A' have the same interpretation in I under v'. Therefore, only one of $A, \neg A'$ is true in I under v', which together with $C, C', \mathsf{ans}(r), \mathsf{ans}(r')$ being false in I under v' forms a contradiction with the assumption that I is a model of both premises of the rule.

Therefore the second binary resolution rule is sound as well. □

By the soundness results of Lemmas 7.7–7.8, Corollaries 7.3–7.4 imply that, when applying the calculus of Figure 7.2 in the saturation-based program synthesis approach of Algorithm 7.1, we construct correct programs.

Example 7.9. We illustrate the use of Algorithm 7.1 with the extended $\mathbb{S}\mathrm{up}$ calculus of Figure 7.2, strengthening our motivation from Section 7.2 with `if−then−else` reasoning. To this end, consider the functional specification over group theory:

$$\forall x, y. \exists z. (x * y \not\simeq y * x \rightarrow z * z \not\simeq \mathsf{e}), \tag{7.11}$$

asserting that, if the group is not commutative, there is an element, the square of which is not e. In addition to the axioms (G1)–(G3) of Figure 7.1, we also use the right identity axiom (G2′) $\forall x.\ x * \mathsf{e} \simeq x$.[3] Based on Algorithm 7.1, we obtain the following derivation of the program for (7.11):

1. $\sigma_1 * \sigma_2 \not\simeq \sigma_2 * \sigma_1 \vee \mathsf{ans}(z)$ [preprocessed specification]
2. $\mathsf{e} \simeq z * z \vee \mathsf{ans}(z)$ [preprocessed specification]
3. $\sigma_1 * \sigma_2 \not\simeq \sigma_2 * \sigma_1$ [answer literal removal 1 (Algorithm 7.1, line 10)]
4. $x * (x * y) \simeq \mathsf{e} * y \vee \mathsf{ans}(x)$ [Sup 2, (G3)]
5. $\mathsf{e} \simeq x * (y * (x * y)) \vee \mathsf{ans}(x * y)$ [Sup (G3), 2]
6. $x * (x * y) \simeq y \vee \mathsf{ans}(x)$ [Sup 4, (G2)]
7. $x * \mathsf{e} \simeq y * (x * y) \vee \mathsf{ans}(\mathtt{if}\ \mathsf{e} \simeq x * (y * (x * y))\ \mathtt{then}\ x\ \mathtt{else}\ x * y)$ [Sup 6, 5]
8. $y * (x * y) \simeq x \vee \mathsf{ans}(\mathtt{if}\ \mathsf{e} \simeq x * (y * (x * y))\ \mathtt{then}\ x\ \mathtt{else}\ x * y)$ [Sup 7, (G2′)]
9. $x * y \simeq y * x \vee \mathsf{ans}(\mathtt{if}\ x * (y * x) \simeq y\ \mathtt{then}\ x\ \mathtt{else}\ \mathtt{if}\ \mathsf{e} \simeq x * (y * (x * y))\ \mathtt{then}\ x\ \mathtt{else}\ x * y)$ [Sup 6, 8]
10. $\mathsf{ans}(\mathtt{if}\ \sigma_1 * (\sigma_2 * \sigma_1) \simeq \sigma_2\ \mathtt{then}\ \sigma_1\ \mathtt{else}\ \mathtt{if}\ \mathsf{e} \simeq \sigma_1 * (\sigma_2 * (\sigma_1 * \sigma_2))\ \mathtt{then}\ \sigma_1\ \mathtt{else}\ \sigma_1 * \sigma_2)$ [BR 9, 3]

[3]We include axiom (G2′) to shorten the presentation of the obtained derivation. The derivation would work also without (G2′).

11. $\square$ [answer literal removal 11 (Algorithm 7.1, line 10)]

The programs with conditions collected during saturation-based program synthesis, in particular corresponding to steps 3. and 11. above, are:

$$\begin{aligned} P_1[x,y] &:= \langle z, x * y \simeq y * x \rangle \\ P_2[x,y] &:= \langle \texttt{if } x * (y * x) \simeq y \texttt{ then } x \texttt{ else } (\texttt{if } \mathsf{e} \simeq x * (y * (x * y)) \texttt{ then } x \texttt{ else } x * y), \\ &\qquad x * y \not\simeq y * x \rangle \end{aligned}$$

Note the variable z, representing an arbitrary witness, in $P_1[x,y]$. An arbitrary value is a correct witness in case $x * y \simeq y * x$ holds, as in this case (7.11) is trivially satisfied. Thus, we do not need to consider the case $x * y \simeq y * x$ separately. Hence, we construct the final program $P[x,y]$ only from $P_2[x,y]$ and obtain:

$$P[x,y] := \texttt{if } x * (y * x) \simeq x \texttt{ then } x \texttt{ else } (\texttt{if } \mathsf{e} \simeq x * (y * (x * y)) \texttt{ then } x \texttt{ else } x * y)$$

$\square$

We conclude this section by illustrating the benefits of computable unifiers.

Example 7.10. Consider the group axioms (G1)–(G3) of Figure 7.1, the additional axioms (G1′) $\forall x.\ x * \mathsf{inv}(x) \simeq \mathsf{e}$ for right inverse and (G2′) $\forall x.\ x * \mathsf{e} \simeq x$ for right identity (symmetric to (G1), (G2)),[4] and the specification

$$\forall x, y. \exists z.\ z * (\mathsf{inv}(x) * \mathsf{inv}(y)) \simeq \mathsf{e}, \tag{7.12}$$

describing the inverse element of $\mathsf{inv}(x) * \mathsf{inv}(y)$. The trivial program derivation for this specification would only have three steps:

1. $\mathsf{e} \not\simeq x * (\mathsf{inv}(\sigma_1) * \mathsf{inv}(\sigma_2)) \vee \mathsf{ans}(x)$ [preprocessed specification]
2. $\mathsf{ans}(\mathsf{inv}(\mathsf{inv}(\sigma_1) * \mathsf{inv}(\sigma_2)))$ [BR (G1), 1]
3. $\square$ [answer literal removal 2]

To disallow the trivial solution, $\mathsf{inv}(\mathsf{inv}(x) * \mathsf{inv}(y))$, we annotate the function symbol inv as uncomputable. Therefore we do not perform step 2 from above, but instead perform binary resolution with the computable unifier $(\{x \mapsto \mathsf{inv}(\mathsf{inv}(\sigma_1) * \mathsf{inv}(\sigma_2))\}, x \not\simeq \mathsf{inv}(\mathsf{inv}(\sigma_1) * \mathsf{inv}(\sigma_2)))$, leading to the following derivation:

1. $\mathsf{e} \not\simeq x * (\mathsf{inv}(\sigma_1) * \mathsf{inv}(\sigma_2)) \vee \mathsf{ans}(x)$ [preprocessed specification]
2. $\mathsf{inv}(\mathsf{inv}(\sigma_1) * \mathsf{inv}(\sigma_2)) \not\simeq x \vee \mathsf{ans}(x)$ [BR (G1), 1]

[4] As in the previous example, we include the symmetric axioms only to shorten the derivation for presentation purposes.

3. $\mathsf{inv}(x) * (x * y) \simeq \mathsf{e} * y$ [Sup (G3), (G1)]
4. $\mathsf{inv}(x) * (x * y) \simeq y$ [Sup 3, (G2)]
5. $\mathsf{inv}(\mathsf{inv}(x)) * \mathsf{e} \simeq x$ [Sup 4, (G1)]
6. $\mathsf{inv}(\mathsf{inv}(x)) \simeq x$ [Sup 5, (G2′)]
7. $\mathsf{e} \simeq x * (y * \mathsf{inv}(x * y))$ [Sup (G1′), (G3)]
8. $x * \mathsf{inv}(y * x) \simeq \mathsf{inv}(x) * \mathsf{e}$ [Sup 4, 7]
9. $\mathsf{inv}(x) \simeq y * \mathsf{inv}(x * y)$ [Sup 8, (G2′)]
10. $\mathsf{inv}(x) * \mathsf{inv}(y) \simeq \mathsf{inv}(y * x)$ [Sup 4, 9]
11. $\mathsf{inv}(\mathsf{inv}(\sigma_2, \sigma_1)) \not\simeq x \lor \mathsf{ans}(x)$ [Sup 10, 2]
12. $\mathsf{ans}(\sigma_2 * \sigma_1)$ [BR 6, 11]
13. $\square$ [answer literal removal 12]

We synthesize the program $P[x, y] := y * x$. Note that there exists a different derivation only using computable unifiers in the form $(\theta, \square)$ (i.e., not using abstraction). $\square$

7.5 Computable Unification with Abstraction

When compared to the $\mathbb{S}$up calculus of Figure 2.2, our extended $\mathbb{S}$up calculus with answer literals from Figure 7.2 uses computable unifiers instead of mgus. To find computable unifiers, we introduce Algorithm 7.2 by extending the standard unification algorithm [Rob65, HV09] and the algorithm for unification with abstraction of [RSV18]. Algorithm 7.2 combines computable unifiers with mgu computation, resulting in the computable unifier $\theta := \mathsf{mgu}_{\mathsf{comp}}(E_1, E_2, E_3)$ to be further used in Figure 7.2.

Algorithm 7.2 modifies a standard unification algorithm to ensure computability of $E_3\theta$. Changes compared with a standard unification algorithm are highlighted. Algorithm 7.2 does not add $s \mapsto t$ to θ if s is a variable in E_3 and t is uncomputable. Instead, if t is $f(t_1, \ldots, t_n)$ where f is computable but not all $t_1, \ldots, t_n$ are computable, we extend θ by $s \mapsto f(x_1, \ldots, x_n)$ and then add equations $x_1 = t_1, \ldots, x_n = t_n$ to the set of equations $\mathcal{E}$ to be processed. Otherwise, f is uncomputable and we perform an abstraction: we consider s and t to be unified under the condition that $s \simeq t$ holds. Therefore we add a constraint $s \not\simeq t$ to the set of literals $\mathcal{D}$ which will be added to any clause invoking the computable unifier. To discharge the literal $s \not\simeq t$, one must prove $s \simeq t$. While s can be later substituted for other terms, as long as we use $\mathsf{mgu}_{\mathsf{comp}}$, s will never be substituted for an uncomputable term. Thus, we conclude the following result.

Theorem 7.11. Let E_1, E_2, E_3 be expressions. Then $(\theta, D) := \mathsf{mgu}_{\mathsf{comp}}(E_1, E_2, E_3)$ is a computable unifier.

Proof. We will denote the subexpression of the expression E at position p by $E|p$.

Algorithm 7.2: Computable Unification with Abstraction

```
function mgu_comp($E_1, E_2, E_3$)
  if $E_3$ is uncomputable then fail
  let $\mathcal{E}$ be a set of equations and $\theta$ be a substitution; $\mathcal{E} := \{E_1 = E_2\}$; $\theta := \{\}$
  let $\mathcal{D}$ be a set of disequalities; $\mathcal{D} := \emptyset$
  repeat
    if $\mathcal{E}$ is empty then
      return $(\theta, D)$ where $D$ is the disjunction of literals in $\mathcal{D}$
    Select an equation $s = t$ in $\mathcal{E}$ and remove it from $\mathcal{E}$
    if $s$ coincides with $t$ then do nothing
    else if $s$ is a variable and $s$ does not occur in $t$ then
      if $s$ does not occur in $E_3$ or $t$ is computable then
        $\theta := \theta \circ \{s \mapsto t\}; \mathcal{E} = \mathcal{E}\{s \mapsto t\}$
      else if $t = f(t_1, \ldots, t_n)$ and $f$ is computable then
        $\theta := \theta \circ \{s \mapsto f(x_1, \ldots, x_n)\};\ \mathcal{E} := \mathcal{E}\{s \mapsto f(x_1, \ldots, x_n)\} \cup \{x_1 = t_1, \ldots, x_n = t_n\}$
          where $x_1, \ldots, x_n$ are fresh variables
      else if $t = f(t_1, \ldots, t_n)$ and $f$ is uncomputable then $\mathcal{D} := \mathcal{D} \cup \{s \not\simeq t\}$
    else if $s$ is a variable and $s$ occurs in $t$ then fail
    else if $t$ is a variable then $\mathcal{E} := \mathcal{E} \cup \{t = s\}$
    else if $s$ and $t$ have different top-level symbols then fail
    else if $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ then
      $\mathcal{E} := \mathcal{E} \cup \{s_1 = t_1, \ldots, s_n = t_n\}$
```

We first prove that (θ, D) is an abstract unifier of E_1, E_2. If $E_1\theta|p'$ and $E_2\theta|p'$ differ, there has to be a position p, where p' is a prefix of p, such that the top-level symbol of $E_1\theta|p$ and $E_2\theta|p$ differs. From the construction of θ follows that for any position p, the subexpressions $E_1\theta|p, E_2\theta|p$ differ in their top-level symbol only if $E_1|p = s$ and $E_2|p = f(t_1, \ldots, t_n)$ (or, symmetrically, $E_1|p = f(t_1, \ldots, t_n)$ and $E_2|p = s$) where s is a variable and f is uncomputable. However, in this case $s \not\simeq f(t_1, \ldots, t_n)$ occurs in D. Therefore, for any interpretation I, any variable assignment v, and any position p', the interpretations of $E_1\theta|p', E_2\theta|p'$ in I under v will either be the same, or $s\theta \not\simeq f(t_1, \ldots, t_n)\theta$ will be true in I under v. Hence, $(D \vee E_1 \simeq E_2)\theta$ is valid, and therefore (θ, D) is an abstract unifier of E_1, E_2.

Next, we prove that $E_3\theta$ is computable. Since the algorithm successfully terminated, E_3 must have been computable (otherwise it would fail). Further, the algorithm only extends the substitution θ by $s \mapsto t$ where t is uncomputable if s does not occur in E_3. Thus, $E_3\theta$ is computable, and hence (θ, D) is a computable unifier. □

7.6 Integrating Synthesis with Splitting in AVATAR

In the final section of this chapter we elaborate on the practical challenges of integrating saturation-based program search with clause splitting, as introduced in Section 2.1.

When using Algorithm 7.1 for program synthesis with (standard) AVATAR to saturate a preprocessed specification (7.5), we may derive a clause $\mathsf{ans}(r[\overline{\sigma}])$ with assertions $C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$. By Theorem 7.2, we then obtain

$$A_1, \ldots, A_n, C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}] \vdash F[\overline{\sigma}, r[\overline{\sigma}]].$$

If $C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$ are computable and ground, then $\langle r[\overline{x}], \bigwedge_{i=1}^m C_i[\overline{x}]\rangle$ is a program with conditions. However, if not all of the assertions $C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$ are computable and ground, then Algorithm 7.1 should continue reasoning with these assertions with the aim of reducing them to computable and ground literals. This, however, is not directly possible in the AVATAR framework.

To preclude this limitation of using AVATAR in saturation-based program synthesis, we modified the AVATAR framework to *only allow splitting over ground computable clauses that do not contain answer literals*. Further, if we derive a clause $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ with AVATAR assertions $C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$, where $C[\overline{\sigma}]$ is ground and computable, we *replace it by the clause* $C[\overline{\sigma}] \vee \bigvee_{i=1}^m \neg C_i[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ *without any assertions*. We then immediately record a program with conditions $\langle r[\overline{x}], \neg C[\overline{x}] \wedge \bigwedge_{i=1}^m C_i[\overline{x}]\rangle$, and replace the clause by $C[\overline{\sigma}] \vee \bigvee_{i=1}^m \neg C_i[\overline{\sigma}]$ (see lines 7-10 of Algorithm 7.1), which may be then further split by AVATAR.

In the following subsections we explain these modifications in detail.

7.6.1 Example without Splitting

We illustrate the pitfalls of using answer literals in AVATAR by exploring Example 2.1 (example originally from [Reg18]).

Example 7.12. We recall the axioms and the specification:

$$\begin{aligned} \text{axioms: } & \mathsf{sunday} \rightarrow \mathsf{workshop}(\mathsf{arcade}) \\ & \mathsf{monday} \rightarrow \mathsf{workshop}(\mathsf{vampire}) \\ & \mathsf{sunday} \vee \mathsf{monday} \\ \text{specification: } & \exists x.\mathsf{workshop}(x) \end{aligned}$$

As explained in Example 2.1, we do not want the symbol $\mathsf{workshop}$ to occur in the target synthesized program, and therefore annotate it as uncomputable. With this annotation, we first straightforwardly synthesize a program for the input without using AVATAR. Note that this derivation differs from the one in Example 2.1 by answer literal removal used in steps 7 and 8 (see line 10 of Algorithm 7.1):

1. $\neg\mathsf{workshop}(x) \vee \mathsf{ans}(x)$ [preprocessed specification]

2.	$\mathsf{sunday} \vee \mathsf{monday}$	[input axiom]
3.	$\neg\mathsf{sunday} \vee \mathsf{workshop}(\mathsf{arcade})$	[input axiom]
4.	$\neg\mathsf{monday} \vee \mathsf{workshop}(\mathsf{vampire})$	[input axiom]
5.	$\neg\mathsf{sunday} \vee \mathsf{ans}(\mathsf{arcade})$	[BR 1, 3]
6.	$\neg\mathsf{monday} \vee \mathsf{ans}(\mathsf{vampire})$	[BR 1, 4]
7.	$\neg\mathsf{sunday}$	[answer literal removal 5]
8.	$\neg\mathsf{monday}$	[answer literal removal 6]
9.	monday	[BR 2, 7]
10.	$\square$	[BR 8, 9]

The programs with conditions recorded in steps 7 and 8 are:

```
if sunday then arcade
if monday then vampire
```

The final program we construct by composing them is the same as in Example 2.1:

```
if sunday then arcade else vampire
```

Note that we do not need to consider the condition `if monday`, because the proof was concluded by deriving $\square$, and hence $\neg\mathsf{sunday}$ combined with the input clauses together implies monday. $\square$

We use this simple problem, which admits a short proof without splitting, to illustrate the issues with using Avatar with answer literals. However, we note that the problems that benefit most from the integration of Avatar and synthesis are more complex, such as the maximum of n variables (for a sufficiently large given constant n):

$$\forall x_1, \ldots, x_n \in \mathbb{Z}.\ \exists y \in \mathbb{Z}.\ \left(\bigwedge_{i=1}^{i\leq n} y \geq_{\mathbb{Z}} x_i \wedge \left(\bigvee_{i=1}^{i\leq n} y \simeq x_i\right)\right) \tag{7.13}$$

7.6.2 Path to Integration

In the process of integrating synthesis with Avatar we dealt with two main questions.

Q1. What would happen if we split clauses containing answer literals? We note that answer literals appear in all clauses only with positive polarity. Hence, if we split a clause containing answer literals such that an answer literal becomes a component and pass it to Avatar, nothing prevents Avatar from finding a model in which all answer literals are true. This model will satisfy all splittable clauses that contain answer literals. Thus, we might only find a proof by refutation if the input axioms without the negated specification were unsatisfiable, since the axioms correspond to the only input clauses without answer literals. Therefore, to avoid answer literals being true in the AVATAR model, we *disallow splitting of clauses that contain answer literals.*

To illustrate our second question, let us take another look at the problem from Example 7.12 using the answer for Q1 from the previous paragraph.

Example 7.13. We search for a proof of the problem from Example 7.12 using AVATAR but without splitting clauses containing answer literals:

1. $\neg\mathsf{workshop}(x) \vee \mathsf{ans}(x)$	[preprocessed specification]
2. $\mathsf{sunday} \vee \mathsf{monday}$	[input axiom]
3. $\neg\mathsf{sunday} \vee \mathsf{workshop}(\mathsf{arcade})$	[input axiom]
4. $\neg\mathsf{monday} \vee \mathsf{workshop}(\mathsf{vampire})$	[input axiom]

AVATAR splits clauses 2-4, denoting the literals sunday, monday, $\mathsf{workshop}(\mathsf{arcade})$, $\mathsf{workshop}(\mathsf{vampire})$ by a, b, c, d, respectively:

$$2\colon\ a \vee b$$
$$3\colon\ \neg a \vee c$$
$$4\colon\ \neg b \vee d$$

Clauses 2-4 now do not participate in the first-order inferences anymore. AVATAR computes a model $\{a, c, d\}$ and introduces the component clauses:

5. $\mathsf{sunday} \leftarrow a$	[component a]
6. $\mathsf{workshop}(\mathsf{arcade}) \leftarrow c$	[component c]
7. $\mathsf{workshop}(\mathsf{vampire}) \leftarrow d$	[component d]

First-order reasoning continues:

8. $\mathsf{ans}(\mathsf{arcade}) \leftarrow c$	[BR 1, 6]
9. $\mathsf{ans}(\mathsf{vampire}) \leftarrow d$	[BR 1, 7]

At this point, there are no more inferences that can be applied. Further, since the clauses containing answer literals (i.e., 8 and 9 also have AVATAR assertions, they are not in the form $C \vee \mathsf{ans}(r)$, where C is ground and computable. Therefore, we cannot apply the answer literal removal steps we used in the AVATAR-less proof, and the proof attempt gets stuck. □

This example leads us to the second question.

Q2. What can we do with a clause derived using assertions and which also contains an answer literal? The Avatar assertions correspond to additional conditions that entail the clause. Hence, a natural way of converting a clause with assertions to a clause without assertions is to add negations of the assertions as literals. I.e., we can convert a clause $C \leftarrow A_1, \ldots, A_n$ to a clause $C \vee \neg A_1 \vee \cdots \vee \neg A_n$.

Example 7.14. Let us try converting the clauses with assertions to assertion-free clauses to continue with our proof from Example 7.13:

10.	$\mathsf{ans}(\mathsf{arcade}) \vee \neg\mathsf{workshop}(\mathsf{arcade})$	[reintroduce assertions of 8]
11.	$\mathsf{ans}(\mathsf{vampire}) \vee \neg\mathsf{workshop}(\mathsf{vampire})$	[reintroduce assertions of 9]

Clauses 10 and 11 are also not in the form $C \vee \mathsf{ans}(r)$ where C is ground and computable, because the symbol $\mathsf{workshop}$ is uncomputable. Therefore, we cannot apply answer literal removal. To make matters even worse, now the first-order reasoning continues by subsuming clauses 10 and 11 by clause 1. Then, there are once again no more first-order inferences that could be applied – we reached saturation with respect to the Avatar model $\{a, c, d\}$, corresponding to $\{\mathsf{sunday}, \mathsf{workshop}(\mathsf{arcade}), \mathsf{workshop}(\mathsf{vampire})\}$. □

Our solution to preclude the situation where we add back assertions only to find that the resulting clause cannot be reasoned with further is to *disallow splitting clauses that are not ground and computable*. Then, all Avatar assertions are ground and computable, and thus if we derive a clause $C \vee \mathsf{ans}(r) \leftarrow A_1, \ldots, A_n$ where C is ground and computable, we can apply answer literal removal. Formally, we do this by introducing a new inference rule

$$\frac{C \vee \mathsf{ans}(r) \leftarrow A_1, \ldots, A_n}{C \vee \neg A_1 \vee \cdots \vee \neg A_n \vee \mathsf{ans}(r)} \text{ (Reintroduce Assertions)},$$

which applies only if C is ground and computable. An application of this rule is always followed by an answer literal removal step, i.e., by recording of the program branch `if` $\neg C \wedge A_1 \wedge \cdots \wedge A_n$ `then` r and replacement of the clause $C \vee \neg A_1 \vee \cdots \vee \neg A_n \vee \mathsf{ans}(r)$ by $C \vee \neg A_1 \vee \cdots \vee \neg A_n$. We note that since the answer literal removal is in fact a simplifying inference (with the side effect of recording a program branch), assertion reintroduction cannot cause looping of the proof search.

Example 7.15. Let us take a look at the proof of the problem from Example 7.12 using Avatar with the constraints and the new rule as described above:

1.	$\neg\mathsf{workshop}(x) \vee \mathsf{ans}(x)$	[preprocessed specification]
2.	$\mathsf{sunday} \vee \mathsf{monday}$	[input axiom]
3.	$\neg\mathsf{sunday} \vee \mathsf{workshop}(\mathsf{arcade})$	[input axiom]
4.	$\neg\mathsf{monday} \vee \mathsf{workshop}(\mathsf{vampire})$	[input axiom]

AVATAR splits the clause 2 (not clauses 3 and 4, since workshop is not computable):

$$2\colon\ a \vee b$$

Clause 2 now does not participate in the first-order inferences anymore. AVATAR computes a model $\{a\}$ and introduces the component clause, and then first-order reasoning continues:

5.	sunday $\leftarrow a$	[component a]
6.	workshop(arcade) $\leftarrow a$	[BR 5, 3]
7.	ans(arcade) $\leftarrow a$	[BR 6, 1]
8.	$\neg$sunday $\vee$ ans(arcade)	[reintroduce assertions 7]
9.	$\neg$sunday	[answer literal removal 8]
10.	$\square \leftarrow a$	[BR 5, 9]
11.	$\neg a$	[AVATAR contradiction 10]

AVATAR recomputes a model $\{b\}$, introduces the component clause for it, and the first-order reasoning continues:

12.	monday $\leftarrow b$	[component b]
13.	workshop(vampire) $\leftarrow b$	[BR 12, 4]
14.	ans(vampire) $\leftarrow b$	[BR 13, 1]
15.	$\neg$monday $\vee$ ans(vampire)	[reintroduce assertions 14]
16.	$\neg$monday	[answer literal removal 15]
17.	$\square \leftarrow b$	[BR 12, 16]
18.	$\neg b$	[AVATAR contradiction 17]

AVATAR tries to recompute the model, but detects that its input clauses are unsatisfiable, which concludes the proof:

19.	$\square$	[AVATAR refutation 2, 11, 18]

Finally, we construct the program from the programs with conditions collected in steps 9 and 16:

```
if sunday then arcade else vampire
```

$\square$

CHAPTER 8

Synthesis of Programs with Recursion

The contributions of this chapter are based on:
Petra Hozzová, Daneshvar Amrollahi, Márton Hajdu, Laura Kovács, Andrei Voronkov, and Eva Maria Wagner. Synthesis of Recursive Programs in Saturation. EasyChair Preprint no. 12145, EasyChair, 2024 [HAH$^+$24], to appear in Proc. of IJCAR *2024*

The synthesis method from the previous chapter worked with the standard superposition calculus. In this chapter we extend it to synthesize recursive programs from proofs using induction by exploiting the correspondence between induction and recursion.

We start this chapter with a motivating example in Section 8.1. We continue in Section 8.2 by giving a high-level overview of the key ideas of our approach. In Section 8.3 we introduce induction axioms, dubbed *magic axioms*, which capture the constructive nature of induction. Next, in Section 8.4 we convert the magic axioms into formulas used by a saturation-based framework to derive programs using recursion over algebraic data types. We also state necessary requirements for the calculus used in saturation and prove correctness of synthesized programs. Then, in Section 8.5 we present an extension of the superposition calculus that fulfills our requirements and advocate for superposition reasoning for recursive function synthesis. Finally, in Section 8.6 we show that our approach, illustrated initially for natural numbers, naturally extends to programs over arbitrary term algebras.

Later in Chapter 9 we describe the implementation of our work in the Vampire prover [KV13] and survey challenging examples it can synthesize.

$$\begin{aligned} \text{axioms:}\quad & \mathsf{half}(0) \simeq 0 && \text{(H1)}\\ & \mathsf{half}(\mathsf{s}(0)) \simeq 0 && \text{(H2)}\\ & \forall x.\ \mathsf{half}(\mathsf{s}(\mathsf{s}(x))) \simeq \mathsf{s}(\mathsf{half}(x)) && \text{(H3)}\\ \text{specification:}\quad & \forall x.\exists y.\ \mathsf{half}(y) \simeq x && (8.1) \end{aligned}$$

Figure 8.1: Axioms of half and the ∀∃-specification for the function computing double.

We note that in this chapter we work with data types, and in particular with natural numbers. Unless indicated differently, quantifiers range over $\mathbb{N}$.

8.1 Motivating Example

Consider the specification (8.1) of Figure 8.1, which describes the inverse of the half function over natural numbers. Given the axiomatization of half (of Figure 2.1, recalled in Figure 8.1), the desired program to synthesize for (8.1) is the recursive function double:

$$\begin{gathered} \mathsf{double}(0) \simeq 0\\ \forall x.\ \mathsf{double}(\mathsf{s}(x)) \simeq \mathsf{s}(\mathsf{s}(\mathsf{double}(x))) \end{gathered} \tag{8.2}$$

The framework from Chapter 7 fails to synthesize a solution of (8.1), as double is a recursive program. To the best of our knowledge, there exists no automated approach supporting recursive function synthesis from functional input-output specifications in full first-order logic.

This chapter provides a solution in this respect by exploiting the constructive nature of induction. Intuitively, each case of an induction axiom tells us how to construct the desired program for the next recursive step using the program for the previous recursive step. We capture this construction recipe contained in the applications of induction in saturation-based proof search, by again utilizing answer literals $\mathsf{ans}(r)$ [Gre69]. When we use an induction axiom in the proof, we introduce a special term into the answer literal, serving for tracking the program corresponding to the induction axiom. As we prove the cases of the induction axiom, we capture their corresponding programs in the answer literal. Finally, when we derive a clause $C \vee \mathsf{ans}(r)$, where C only contains symbols allowed in a program, we convert the special tracker terms from r into recursive functions, and then, similarly to Chapter 7, obtain a program for the initial specification conditioned on $\neg C$.

8.2 Saturation with Induction in Constructive Logic

In this section we summarize the key challenges our work resolves towards recursive synthesis in saturation. The idea of extracting programs from proofs originates from results in constructive (intuitionistic) logic, starting with Kleene's realizability [Kle45]. In constructive logic, provability of a formula $\forall\overline{x}.\exists y.F[\overline{x}, y]$ implies that there is an algorithm which, given values for $\overline{x}$, outputs a value for y satisfying $F[\overline{x}, y]$ (for $\overline{x}, y$ of any sorts).

We note that the structural induction axiom (2.2) over natural numbers has computational content, as follows. The program r for $\forall x.G[x]$ can be built from a program r_0 for $G[0]$ and a program r_s for $\forall y.(G[y] \to G[\mathsf{s}(y)])$ as:

$$r(0) \simeq r_0$$
$$r(\mathsf{s}(y)) \simeq r_\mathsf{s}(r(y))$$

For this to be useful, we need to first prove $G[0]$, then prove $\forall x.(G[y] \to G[\mathsf{s}(y)])$, and then use the induction axiom to derive $\forall x.G[x]$. Such an approach towards constructing programs does not however work in saturation-based theorem proving, as saturation does not reduce goals to subgoals [Bon99]. Rather, as explained in Section 2.3, we add the induction axiom as a theory lemma to the proof search and continue saturation. Thus, we do not have proofs of either $G[0]$ or $\forall y.(G[y] \to G[\mathsf{s}(y)])$. Constructing programs during saturation becomes even more complex when using answer literals, because clauses generated during saturation may contain these literals. For example, if we try to extract a proof of $G[0]$, we may find a proof with an answer literal in it.

To capture the constructive nature of induction and address the above challenges of program synthesis in saturation, we use the following trick. We modify the induction axiom so that it indirectly stores information about the programs for $G[0]$ and $\forall y.(G[y] \to G[\mathsf{s}(y)])$. To do this, instead of adding the induction axiom (2.2), in Section 8.3 we add what we call a *magic axiom for* (2.2), where G has an additional argument for storing the program. In Section 8.4 we further convert our magic axioms into formulas to be used to derive recursive programs in saturation.

8.3 Induction with Magic Formulas

We first present our approach to *proving* formulas with a free variable by induction. We further extend this approach to *synthesis* in Section 8.4. While our approach works the same way with arbitrary term algebras, for the sake of clarity we first introduce our work for natural numbers and then for general term algebras in Section 8.6.

Let $G[t, x]$ be a formula with a single free variable $x : \alpha$ containing a term $t : \mathbb{N}$. We use the following *magic axiom*:

$$\Big(\exists v_0 \in \alpha.\ G[0, v_0] \land \forall y.(\exists w \in \alpha.\ G[y, w] \to \exists v_\mathsf{s} \in \alpha.\ G[\mathsf{s}(y), v_\mathsf{s}])\Big) \to \forall z.\exists x \in \alpha.\ G[z, x] \quad (8.3)$$

Note that all magic axioms are valid, as they are instances of the structural induction schema (2.2) with the quantified formula $\exists x \in \alpha.G[t, x]$ in place of $G[t]$. The magicalness of (8.3) stems from its simple, yet powerful expressiveness: when used in proof search, the variables v_0, v_s in the antecedent capture the programs for the base and step cases, allowing us to construct a program for x in the consequent.

Using axiom (8.3), we introduce the following variant of the Ind rule:

$$\frac{\overline{L}[t,x] \vee C}{\big(\exists v_0 \in \alpha.L[0,v_0] \wedge \forall y.(\exists w \in \alpha.L[y,w] \rightarrow \exists v_{\mathsf{s}} \in \alpha.L[\mathsf{s}(y),v_{\mathsf{s}}])\big) \rightarrow \forall z.\exists x \in \alpha.L[z,x]} \;(\mathsf{MagInd})$$

where the only free variable of $L[t,x]$ is x and C does not contain x.

Example 8.1. Consider the specification (8.1) from Figure 8.1. To prove it using superposition, and not yet synthesize the function satisfying (8.1), we use the following magic axiom:

$$\big(\exists v_0.\mathsf{half}(v_0)\simeq 0 \wedge \forall y.(\exists w.\mathsf{half}(w)\simeq y \rightarrow \exists v_{\mathsf{s}}.\mathsf{half}(v_{\mathsf{s}})\simeq \mathsf{s}(y))\big) \rightarrow \forall z.\exists x.\mathsf{half}(x)\simeq z \quad (8.4)$$

To use (8.4) in saturation, we clausify it and skolemize the variables y, w, x as $\sigma_y, \sigma_w, \sigma_x(z)$, respectively. The following is a refutational proof of (8.1):

1.	$\mathsf{half}(y) \not\simeq \sigma$	[preprocessed specification]
2.	$\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(\sigma_w) \simeq \sigma_y \vee \mathsf{half}(\sigma_x(z)) \simeq z$	[MagInd with (8.4)]
3.	$\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(v_{\mathsf{s}}) \not\simeq \mathsf{s}(\sigma_y) \vee \mathsf{half}(\sigma_x(z)) \simeq z$	[MagInd with (8.4)]
4.	$\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(\sigma_w) \simeq \sigma_y$	[BR 1, 2]
5.	$\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(v_{\mathsf{s}}) \not\simeq \mathsf{s}(\sigma_y)$	[BR 1, 3]
6.	$\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(v_{\mathsf{s}}) \not\simeq \mathsf{s}(\mathsf{half}(\sigma_w))$	[Sup 4, 5]
7.	$\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(v_{\mathsf{s}}) \not\simeq \mathsf{half}(\mathsf{s}(\mathsf{s}(\sigma_w)))$	[Sup (H3), 6]
8.	$\mathsf{half}(v_0) \not\simeq 0$	[ER 7]
9.	$\square$	[BR 8,(H2)]

Hence, the magic axiom (8.3) is sufficient to prove (8.1). However, (8.3) does not suffice to synthesize the program for (8.1) from the above proof. Similarly as in Chapter 7, for synthesis we would use

$$\mathsf{half}(y) \not\simeq \sigma \vee \mathsf{ans}(y) \quad (8.5)$$

instead of clause 1 and obtain a derivation similar to the one above, but with the answer literal $\mathsf{ans}(\sigma_x(\sigma))$. As σ_x is a fresh skolem function, it is uncomputable and not allowed in answer literals. Therefore, simply following the approach of Chapter 7 fails to synthesize a recursive program from the proof of (8.1). We address the challenge of program construction for the skolem function σ_x in the following section. $\square$

8.4 Programs with Primitive Recursion

We now construct recursive programs for proofs using induction over natural numbers (8.3). In Section 8.6 we generalize the approach to any term algebra.

As mentioned in Section 8.2, the antecedent of the induction axiom gives us a recipe for constructing the program for the consequent. To capture this dependence of the consequent program x on the antecedent programs v_0, v_s, we convert the magic axiom (8.3) to its equivalent prenex normal form, where $\forall v_0, v_\mathsf{s}$ precedes $\exists x$:

$$\exists y.\ \exists w \in \alpha.\ \forall v_0, v_\mathsf{s} \in \alpha.\ \forall z.\ \exists x \in \alpha.\ \Big((G[\mathsf{0}, v_0] \wedge (G[y, w] \rightarrow G[\mathsf{s}(y), v_\mathsf{s}])) \rightarrow G[z, x] \Big) \tag{8.6}$$

We next define a recursive operator to be used for constructing programs.

Definition 8.2 (Primitive Recursion Operator)**.** Let $f_1 : \alpha$, and $f_2 : \mathbb{N} \times \alpha \rightarrow \alpha$. The *primitive recursion operator* R *for natural numbers and* α is:

$$\begin{aligned} \mathsf{R}(f_1, f_2)(0) &\simeq f_1 \\ \mathsf{R}(f_1, f_2)(\mathsf{s}(y)) &\simeq f_2(y, \mathsf{R}(f_1, f_2)(y)) \end{aligned}$$

□

Lemma 8.3 (Recursive Witness)**.** The expression $\mathsf{R}(v_0, \lambda y, w.v_\mathsf{s})(z)$ is a witness for the variable x in (8.6).

Proof. Let us consider the following formula obtained by replacing x in (8.6) by $\mathsf{R}(v_0, \lambda y, w.v_\mathsf{s})(z)$:

$$\exists y.\ \exists w \in \alpha.\ \forall v_0, v_\mathsf{s} \in \alpha.\ \forall z.\ \Big((G[\mathsf{0}, v_0] \wedge (G[y, w] \rightarrow G[\mathsf{s}(y), v_\mathsf{s}])) \rightarrow G[z, \mathsf{R}(v_0, \lambda y, w.v_\mathsf{s})(z)] \Big) \tag{8.7}$$

We will prove that every interpretation I is a model of (8.7).

By contradiction, let us assume that (8.7) is false in I. That would mean that for all values a, b and an interpretation $I\{y \mapsto a, w \mapsto b\}$, there exists an extension $J_{a,b}$ of $I\{y \mapsto a, w \mapsto b\}$ by $\{v_0 \mapsto o_0, v_\mathsf{s} \mapsto o_\mathsf{s}, z \mapsto o_z\}$ for some values o_0, o_s, o_z which depend on a, b, such that:

$$\Big((G[\mathsf{0}, v_0] \wedge (G[y, w] \rightarrow G[\mathsf{s}(y), v_\mathsf{s}])) \rightarrow G[z, \mathsf{R}(v_0, \lambda y, w.v_\mathsf{s})(z)] \Big)^{J_{a,b}} = \bot \tag{8.8}$$

This means that for any values a, b:[1]

$$G^{J_{a,b}}[\mathsf{0}, v_0^{J_{a,b}}] = \top \tag{8.9}$$

$$G^{J_{a,b}}[a, b] = \bot \quad \text{or} \quad G^{J_{a,b}}[\mathsf{s}(a), v_\mathsf{s}^{J_{a,b}}] = \top \tag{8.10}$$

$$G^{J_{a,b}}[z^{J_{a,b}}, \mathsf{R}^{J_{a,b}}(v_0^{J_{a,b}}, (\lambda y, w.v_\mathsf{s})^{J_{a,b}})(z^{J_{a,b}})] = \bot \tag{8.11}$$

[1]In the following, we write $v_0^{J_{a,b}}, v_\mathsf{s}^{J_{a,b}}$ instead of o_0, o_s to emphasize the dependence of the interpretation of v_0, v_s on the values a, b.

Since the operator R has a fixed interpretation and since the variables $y, w, v_0, v_\mathsf{s}, z$ do not occur in $G[x_1, x_2]$ (where x_1, x_2 are fresh), from (8.9)-(8.11) we obtain for any values a, b:

$$G^I[\mathsf{0}, v_0^{J_{a,b}}] = \top \tag{8.12}$$

$$G^I[a, b] = \bot \quad \text{or} \quad G^I[\mathsf{s}(a), v_\mathsf{s}^{J_{a,b}}] = \top \tag{8.13}$$

$$G^I[z^{J_{a,b}}, \mathsf{R}^I(v_0^{J_{a,b}}, (\lambda y, w.v_\mathsf{s})^{J_{a,b}})(z^{J_{a,b}})] = \bot \tag{8.14}$$

By definition $(\lambda y, w.v_\mathsf{s})^{J_{a,b}} = f_{a,b}$ where for any o_1, o_2: $f_{a,b}(o_1, o_2) = v_\mathsf{s}^{J_{a,b}\{y \mapsto o_1, w \mapsto o_2\}}$. Since $J_{a,b}\{y \mapsto o_1, w \mapsto o_2\} = J_{o_1,o_2}$, the function $f_{a,b}$ actually does not depend on a, b and thus we define $f = f_{a,b}$ and obtain:

$$f(o_1, o_2) = v_\mathsf{s}^{J_{o_1,o_2}} \tag{8.15}$$

Using (8.15) we obtain from (8.13) and (8.14) for any a, b:

$$G^I[a, b] = \bot \quad \text{or} \quad G^I[\mathsf{s}(a), f(a, b)] = \top \tag{8.16}$$

$$G^I[z^{J_{a,b}}, \mathsf{R}^I(v_0^{J_{a,b}}, f)(z^{J_{a,b}})] = \bot \tag{8.17}$$

We now consider (8.17) for arbitrarily fixed values $a := o_y, b := o_w$. As $o_z = z^{J_{o_y,o_w}}, o_0 = v_0^{J_{o_y,o_w}}$, we obtain:

$$G^I[o_z, \mathsf{R}^I(o_0, f)(o_z)] = \bot \tag{8.18}$$

Assume there is a smallest value of o_z such that (8.18) holds. Either this value is $\mathsf{0}$, or a successor of some o, i.e., $o_z = \mathsf{s}(o)$:

1. If $o_z = \mathsf{0}$, then $\mathsf{R}^I(o_0, f)(o_z) = \mathsf{R}^I(o_0, f)(\mathsf{0})$ is by definition of R equal to o_0. Therefore,

$$\bot \overset{(8.18)}{=} G^I[\mathsf{0}, \mathsf{R}^I(o_0, f)(\mathsf{0})] = G^I[\mathsf{0}, o_0] \overset{(8.12)\text{ with } a:=o_y, b:=o_w}{=} \top.$$

   This is a contradiction and thus it has to be the second case:

2. $o_z = \mathsf{s}(o)$, therefore from (8.18):

$$G^I[\mathsf{s}(o), \mathsf{R}^I(o_0, f)(\mathsf{s}(o))] = \bot$$

   By definition of R we have $\mathsf{R}^I(o_0, f)(\mathsf{s}(o)) = f(o, \mathsf{R}^I(o_0, f)(o))$, and thus:

$$G^I[\mathsf{s}(o), f(o, \mathsf{R}^I(o_0, f)(o))] = \bot \tag{8.19}$$

   From (8.16) with $a := o, b := \mathsf{R}^I(o_0, f)(o)$ we obtain:

$$G^I[o, \mathsf{R}^I(o_0, f)(o)] = \bot \quad \text{or} \quad G^I[\mathsf{s}(o), f(o, \mathsf{R}^I(o_0, f)(o))] = \top$$

From that and (8.19) we get:

$$G^I[o, \mathsf{R}^I(o_0, f)(o)] = \bot$$

That is, o satisfies (8.18), which contradicts the assumption that $\mathsf{s}(o)$ is the smallest value satisfying (8.18).

Therefore, there is no value o_z satisfying (8.18). Thus, since the argument above works for arbitrary $o_y = a, o_w = b$, the formula from (8.8) cannot be false in any $J_{a,b}$. This means that (8.7) cannot be false in any I, meaning that it is valid and $\mathsf{R}(v_0, \lambda y, w.v_\mathsf{s})(z)$ is indeed a witness for the variable x in (8.6). □

Lemma 8.3 ensures that we can construct a program for the consequent of the magic axiom given programs for the base case and the step case. We next integrate this construction into our synthesis framework using answer literals. For that we take a close look at skolemization of induction axiom (8.6), and define skolem symbols for the variable x, capturing the recursive program.

Definition 8.4 (rec-Symbols)**.** Consider formulas $G[t, x]$ with a single free variable $x : \alpha$ containing a term $t : \mathbb{N}$. For each such formula, we introduce a distinct computable function symbol $\mathsf{rec}_{G[t,x]} : \alpha \times \alpha \times \mathbb{N} \to \alpha$. We will refer to such symbols $\mathsf{rec}_{G[t,x]}$ as rec-*symbols*. When the formula $G[t, x]$ is clear from the context or unimportant for the context, we will simply write rec instead of $\mathsf{rec}_{G[t,x]}$. □

A term with a rec-symbol as the top-level functor is called a rec-*term*.

Definition 8.5 (Magic Formula)**.** *The magic formula for* $G[t, x]$ is:

$$\forall v_0, v_\mathsf{s} \in \alpha.\ \forall z.\ \Big(\big(G[0, v_0] \land (G[\sigma_y, \sigma_w] \to G[\mathsf{s}(\sigma_y), v_\mathsf{s}])\big) \to G[z, \mathsf{rec}_{G[t,x]}(v_0, v_\mathsf{s}, z)]\Big) \quad (8.20)$$

□

It is easy to see that magic formula (8.20) is obtained by skolemizing the prenex normal form of magic axiom (8.6), where we replace the variables y, w by fresh constants σ_y, σ_w, and the variable x by a fresh $\mathsf{rec}_{G[t,x]}$-symbol. The constants σ_y, σ_w introduced in (8.20) are said to be *associated with the* $\mathsf{rec}_{G[t,x]}$*-term.* An occurrence of any skolem constant σ_y, σ_w is considered computable if it is an occurrence in the second argument of a $\mathsf{rec}_{G[t,x]}$-term which it is associated with.

We introduce additional requirements for reasoning with rec-terms to ensure that they always represent the recursive function to be synthesized.

Definition 8.6 (rec-Compliance)**.** An inference system $\mathcal{I}$ is rec-*compliant* if:

1. $\mathcal{I}$ only introduces rec-terms in the instances of the magic formula (8.20),

2. $\mathcal{I}$ does not introduce uncomputable symbols into arguments of rec-terms in clauses it derives.

$\square$

Using a rec-compliant inference system $\mathcal{I}$, we derive clauses containing rec-terms. These terms correspond to functions constructed using the operator R.

Definition 8.7 (Recursive Function Term)**.** Let σ_y, σ_w be associated with $\mathsf{rec}(s_1, s_2, t)$. Then we call the term $\mathsf{R}(s_1, \lambda\sigma_y, \sigma_w.s_2)(t)$ the *recursive function term corresponding to* $\mathsf{rec}(s_1, s_2, t)$. $\square$

For a term r, we denote by r^{R} the expression obtained from r by iteratively replacing all rec-terms by their corresponding recursive function terms, starting from the innermost ones. Similarly, formula F^{R} denotes the formula F in which we replace all rec-terms by their corresponding recursive function terms.

Lemma 8.8 (Recursive Witness for Magic Formulas)**.** Consider the formula obtained from (8.20) by replacing $\mathsf{rec}_{G[t,x]}(v_0, v_\mathsf{s}, z)$ by its corresponding recursive function term $\mathsf{R}(v_0, \lambda\sigma_y, \sigma_w.v_\mathsf{s})(z)$:

$$\forall v_0, v_\mathsf{s} \in \alpha.\forall z.\Big((G[\mathsf{0}, v_0] \wedge (G[\sigma_y, \sigma_w] \rightarrow G[\mathsf{s}(\sigma_y), v_\mathsf{s}])) \rightarrow G[z, \mathsf{R}(v_0, \lambda\sigma_y, \sigma_w.v_\mathsf{s})(z)]\Big) \quad (8.21)$$

For every interpretation I, there exists a mapping of skolem constants to values $\{\sigma_y \mapsto o_y, \sigma_w \mapsto o_w\}$ such that I extended by this mapping is a model of (8.21). As a consequence, formula (8.21) is satisfiable.

Proof. The lemma immediately follows from the fact that formula (8.21) is a skolemization of

$$\exists y.\exists w \in \alpha.\forall v_0, v_\mathsf{s} \in \alpha.\forall z.\Big((G[\mathsf{0}, v_0] \wedge (G[y, w] \rightarrow G[\mathsf{s}(y), v_\mathsf{s}])) \rightarrow G[z, \mathsf{R}(v_0, \lambda y, w.v_\mathsf{s})(z)]\Big),$$

which is by Lemma 8.3 valid. $\square$

Lemma 8.8 implies that we can use formula (8.21) instead of (8.20) in derivation, while preserving the soundness of the derivations. The following theorem, based on Theorem 7.2, is the key theorem showing the soundness of our approach to recursive program derivation.

Theorem 8.9 (Semantics of Clauses with Answer Literals and rec-terms)**.** Let $C_1, \ldots, C_m$ be clauses and F a formula containing no answer literals and no rec-symbols. Let C be a clause containing no answer literals. Let $M_1, \ldots, M_l$ be magic formulas. Assume that using a sound rec-compliant inference system $\mathcal{I}$, we derive $C \vee \mathsf{ans}(r[\overline{\sigma}])$, where $r[\overline{\sigma}]$ is computable, from the set of clauses

$$\{\ C_1, \ldots, C_m,\ \ M_1, \ldots, M_l,\ \ \mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee \mathsf{ans}(y))\ \}.$$

Then

$$M_1^{\mathsf{R}}, \ldots, M_l^{\mathsf{R}}, C_1, \ldots, C_m \vdash C^{\mathsf{R}}, F[\overline{\sigma}, r^{\mathsf{R}}[\overline{\sigma}]].$$

That is, under the assumptions $M_1^{\mathsf{R}}, \ldots, M_l^{\mathsf{R}}, C_1, \ldots, C_m, \neg C^{\mathsf{R}}$, the computable expression $r^{\mathsf{R}}[\overline{x}]$ is a witness for y in $\forall \overline{x}.\exists y.F[\overline{x}, y]$.[2]

Proof. The proof mirrors the proof of Theorem 7.2.

We consider the calculus that was used for deriving $C \vee \mathsf{ans}(r[\overline{\sigma}])$, but with lifted ordering and selection conditions, i.e., we allow application of the rules regardless of the term ordering and of which literals are selected. Since the soundness of the calculus does not depend on these side conditions, the calculus without the conditions is sound as well. Now, since ans is uninterpreted, we can replace $\mathsf{ans}(y)$ by $y \not\simeq r^{\mathsf{R}}[\overline{\sigma}]$. Further, since also all rec-symbols are uninterpreted, we can also replace each $\mathsf{rec}(\overline{v}, z)$ in each induction formula M_i by its corresponding recursive function term. Since the calculus is rec-compliant, all rec-terms were introduced by the induction formulas $M_1, \ldots, M_l$, and therefore after the replacements we obtain a derivation of $C^{\mathsf{R}} \vee r^{\mathsf{R}}[\overline{\sigma}] \not\simeq r^{\mathsf{R}}[\overline{\sigma}]$ from $\forall y.\mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r^{\mathsf{R}}[\overline{\sigma}]), M_1^{\mathsf{R}}, \ldots, M_l^{\mathsf{R}}, C_1, \ldots, C_m$ using the calculus without the conditions.[3]

We want to show that

$$\bigwedge_{i=1}^{l} M_i^{\mathsf{R}} \wedge \bigwedge_{i=1}^{m} C_i \rightarrow C^{\mathsf{R}} \vee F[\overline{\sigma}, r^{\mathsf{R}}[\overline{\sigma}]] \tag{8.22}$$

is valid. Hence, we need to show that in each interpretation, in which the antecedent is true, also the consequent is true. Let us consider such an interpretation I. We distinguish two cases:

1. First, assume that $\forall y.\mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r^{\mathsf{R}}[\overline{\sigma}])$ is true in I. Then since all assumptions from which we derived $C^{\mathsf{R}} \vee r^{\mathsf{R}}[\overline{\sigma}] \not\simeq r^{\mathsf{R}}[\overline{\sigma}]$ are true in I and since the inference system is sound, also $C^{\mathsf{R}} \vee r^{\mathsf{R}}[\overline{\sigma}] \not\simeq r^{\mathsf{R}}[\overline{\sigma}]$ is true. That clause is equivalent to C^{R}, hence C^{R} is true, which makes the consequent of (8.22) true.

2. Second, assume that $\forall y.\mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r^{\mathsf{R}}[\overline{\sigma}])$ is false in I. Then its negation, $\neg\forall y.\mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r^{\mathsf{R}}[\overline{\sigma}])$, equivalent to $\exists y.(F[\overline{\sigma}, y] \wedge y \simeq r^{\mathsf{R}}[\overline{\sigma}])$, equivalent to $F[\overline{\sigma}, r^{\mathsf{R}}[\overline{\sigma}]]$ must be true in I. Hence, the consequent of (8.22) is true also in this case.

Therefore (8.22) is valid. Since $\overline{\sigma}$ are fresh uninterpreted constants, we obtain that $\bigwedge_{i=1}^{l} M_i^{\mathsf{R}} \wedge \bigwedge_{i=1}^{m} C_i \rightarrow C^{\mathsf{R}} \vee F[\overline{x}, r^{\mathsf{R}}[\overline{x}]]$ is valid too, and hence $r^{\mathsf{R}}[\overline{x}]$ is a witness for $\forall \overline{x}.\exists y.F[\overline{x}, y]$ under the assumptions $M_1^{\mathsf{R}}, \ldots, M_l^{\mathsf{R}}, C_1, \ldots, C_m, \neg C^{\mathsf{R}}$.

[2]for $\overline{x}, y$ of any sorts

[3]The derivation might not have been possible in the calculus with the ordering and selection conditions due to replacing the positive literal $\mathsf{ans}(y)$ with the negative literal $y \not\simeq r^{\mathsf{R}}[\overline{\sigma}]$ containing different symbols, and replacing rec-terms by terms with R and λ.

Finally, note that since $r[\overline{\sigma}]$ is computable, so is $r[\overline{x}]$. The only skolem constants $r[\overline{x}]$ contains are skolem constants within the respective arguments of rec-terms they are associated with. Since $r^{\mathsf{R}}[\overline{x}]$ lambda-binds exactly those skolem constants from the rec-terms, we have that $r^{\mathsf{R}}[\overline{x}]$ is computable too. □

Based on Theorem 8.9, if the CNF of $A_1, \dots, A_n$ is among $C_1, \dots, C_m$, then $r^{\mathsf{R}}[\overline{x}]$ is a witness for y in (7.4) under the assumptions $M_1^{\mathsf{R}}, \dots, M_l^{\mathsf{R}}, C_1, \dots, C_m, \neg C^{\mathsf{R}}$. We note that we consider (7.4) for $\overline{x}, y$ of any sorts. The following theorem, based on Corollary 7.3, ensures that we can construct recursive programs with conditions.

Theorem 8.10 (Recursive Programs with Conditions)**.** Let $r[\overline{\sigma}]$ be a computable term, and $C[\overline{\sigma}], C_1[\overline{\sigma}], \dots, C_m[\overline{\sigma}]$ be ground computable clauses containing no answer literals and no rec-symbols. Assume that using a sound rec-compliant inference system $\mathcal{I}$, we derive the clause $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ from the CNF of

$$\{\ A_1, \dots, A_n,\ C_1[\overline{\sigma}], \dots, C_m[\overline{\sigma}],\ M_1, \dots, M_l,\ \neg F[\overline{\sigma}, y] \vee \mathsf{ans}(y)\ \}$$

where $M_1, \dots, M_l$ are magic formulas. Then,

$$\langle r^{\mathsf{R}}[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}] \rangle$$

is a program with conditions for (7.4).

Proof. From Theorem 8.9 follows that

$$\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{l} M_i^{\mathsf{R}} \wedge \bigwedge_{i=1}^{m} C_i[\overline{\sigma}] \rightarrow C^{\mathsf{R}}[\overline{\sigma}] \vee F[\overline{\sigma}, r^{\mathsf{R}}[\overline{\sigma}]] \tag{8.23}$$

is valid. Since $C[\overline{\sigma}]$ does not contain any rec-terms, $C^{\mathsf{R}}[\overline{\sigma}] = C[\overline{\sigma}]$. We can therefore equivalently rewrite (8.23) as

$$\bigwedge_{i=1}^{l} M_i^{\mathsf{R}} \rightarrow (\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{m} C_i[\overline{\sigma}] \rightarrow C[\overline{\sigma}] \vee F[\overline{\sigma}, r^{\mathsf{R}}[\overline{\sigma}]]). \tag{8.24}$$

Let us consider an arbitrary interpretation I of

$$\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{m} C_i[\overline{\sigma}] \rightarrow C[\overline{\sigma}] \vee F[\overline{\sigma}, r^{\mathsf{R}}[\overline{\sigma}]]. \tag{8.25}$$

From Lemma 8.8 follows that we can extend I to I' which is a model of $\bigwedge_{i=1}^{l} M_i^{\mathsf{R}}$ by choosing suitable values for skolem constants in each M_i (each of these skolems only occurs in one M_i, and they do not occur in (8.25)). Since (8.24) is valid and $\bigwedge_{i=1}^{l} M_i^{\mathsf{R}}$ is true in I', also (8.25) has to be true in I'. However, since (8.25) does not contain any of those skolem constants by which I was extended to I', we get that (8.25) is also true in I. Therefore (8.25) is valid.

Next, since $\overline{\sigma}$ are fresh uninterpreted constants, we obtain that the formula $\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{m} C_i[\overline{x}] \rightarrow C[\overline{x}] \vee F[\overline{x}, r^{\mathsf{R}}[\overline{x}]]$ is valid as well, and this formula is equivalent to $\bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}] \rightarrow (\bigwedge_{i=1}^{n} A_i \rightarrow F[\overline{x}, r^{\mathsf{R}}[\overline{x}]])$. Therefore $\langle r^{\mathsf{R}}[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}]\rangle$ is a program with conditions for $A_1 \wedge \ldots \wedge A_n \rightarrow \forall \overline{x}. \exists y. F[\overline{x}, y]$. □

From Theorem 8.10 we obtain the following key result on program synthesis, analogous to Corollary 7.4.

Theorem 8.11 (Recursive Program Synthesis)**.** Let $P_1[\overline{x}], \ldots, P_k[\overline{x}]$, where $P_i[\overline{x}] = \langle r_i^{\mathsf{R}}[\overline{x}], \bigwedge_{j=1}^{i-1} C_j[\overline{x}] \wedge \neg C_i[\overline{x}]\rangle$, be programs with conditions for (7.4), such that $\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{k} C_i[\overline{x}]$ is unsatisfiable. Then the program $P[\overline{x}]$ defined as

$$\begin{aligned} P[\overline{x}] := \ & \texttt{if } \neg C_1[\overline{x}] \texttt{ then } r_1^{\mathsf{R}}[\overline{x}] \\ & \texttt{else if } \neg C_2[\overline{x}] \texttt{ then } r_2^{\mathsf{R}}[\overline{x}] \\ & \quad \ldots \\ & \texttt{else if } \neg C_{k-1}[\overline{x}] \texttt{ then } r_{k-1}^{\mathsf{R}}[\overline{x}] \\ & \texttt{else } r_k^{\mathsf{R}}[\overline{x}], \end{aligned}$$

is a program for (7.4).

Proof. The proof is the same as for Corollary 7.4. □

8.5 Recursive Synthesis in Saturation

This section integrates the proving and synthesis steps of Sections 8.3–8.4 into saturation. The crux of our approach is that instead of adding standard induction formulas to the search space, we add magic formulas.

Theorems 8.10–8.11 imply that, to derive recursive programs, we can use any rec-compliant calculus, as long as the calculus supports derivation of clauses $C \vee \mathsf{ans}(r)$, where r is computable and C is ground, computable, and contains neither rec-terms nor answer literals. In our work we rely on the extended $\mathbb{S}$up calculus of Figure 7.2, which we:

1. further extend by adding magic formulas alongside standard induction formulas when using MagInd,
2. make rec-compliant by disallowing inferences containing uncomputable rec-terms,
3. extend by adding more complex rules for introducing conditions into rec-terms. We display the new rules in Figure 8.2. These rules are useful when the BR and Sup rules from the left-hand column of Figure 7.2 do not apply, because the condition

Superposition (Sup):

$$\frac{\underline{s \simeq t} \vee C \vee \mathsf{ans}(r[\mathsf{rec}(r_0, r_\mathsf{s}, r'')]) \quad \underline{L[s']} \vee C' \vee \mathsf{ans}(r'[\mathsf{rec}(r_0, r'_\mathsf{s}, r'')])}{(\underline{D} \vee L[t] \vee C \vee C' \vee \mathsf{ans}(r[\mathsf{rec}(r_0, \mathtt{if}\ s \simeq t\ \mathtt{then}\ r'_\mathsf{s}\ \mathtt{else}\ r_\mathsf{s}, r'')]))\theta}$$

$$\frac{\underline{s \simeq t} \vee C \vee \mathsf{ans}(r[\mathsf{rec}(r_0, r_\mathsf{s}, r'')]) \quad \underline{u[s'] \not\simeq u'} \vee C' \vee \mathsf{ans}(r'[\mathsf{rec}(r_0, r'_\mathsf{s}, r'')])}{(\underline{D} \vee u[t] \not\simeq u' \vee C \vee C' \vee \mathsf{ans}(r[\mathsf{rec}(r_0, \mathtt{if}\ s \simeq t\ \mathtt{then}\ r'_\mathsf{s}\ \mathtt{else}\ r_\mathsf{s}, r'')]))\theta}$$

$$\frac{\underline{s \simeq t} \vee C \vee \mathsf{ans}(r[\mathsf{rec}(r_0, r_\mathsf{s}, r'')]) \quad \underline{u[s'] \simeq u'} \vee C' \vee \mathsf{ans}(r'[\mathsf{rec}(r_0, r'_\mathsf{s}, r'')])}{(\underline{D} \vee u[t] \simeq u' \vee C \vee C' \vee \mathsf{ans}(r[\mathsf{rec}(r_0, \mathtt{if}\ s \simeq t\ \mathtt{then}\ r'_\mathsf{s}\ \mathtt{else}\ r_\mathsf{s}, r'')]))\theta}$$

where (θ, D) is a computable unifier of s, s' and r, r', both w.r.t. $r[\mathsf{rec}(r_0,$ $\mathtt{if}\ s \simeq t\ \mathtt{then}\ r'_\mathsf{s}\ \mathtt{else}\ r_\mathsf{s}, r'')]$; (first rule only) $L[s']$ is not an equality literal; and (second and third rule only) $u'\theta \not\succeq u[s']\theta$.

Binary resolution (BR):

$$\frac{\underline{A} \vee C \vee \mathsf{ans}(r[\mathsf{rec}(r_0, r_\mathsf{s}, r'')]) \quad \neg\underline{A'} \vee C' \vee \mathsf{ans}(r'[\mathsf{rec}(r_0, r'_\mathsf{s}, r'')])}{(\underline{D} \vee C \vee C' \vee \mathsf{ans}(r[\mathsf{rec}(r_0, \mathtt{if}\ A\ \mathtt{then}\ r'_\mathsf{s}\ \mathtt{else}\ r_\mathsf{s}, r'')]))\theta}$$

where (θ, D) is a computable unifier of A, A' and r, r', both w.r.t. $r[\mathsf{rec}(r_0, \mathtt{if}\ A\ \mathtt{then}\ r'_\mathsf{s}\ \mathtt{else}\ r_\mathsf{s}, r'')]$.

Figure 8.2: Rules of the extended superposition calculus $\mathbb{S}$up for reasoning with answer literals with rec-terms. The underlined literals are selected.

of the if − then − else is not computable outside of the second argument of the rec-term – i.e., when the condition contains skolem constants associated with the rec-term. Applying these new rules results in conditions that are local to the recursive branch of the synthesized recursive function, and use the argument of the recursive call, or the result of the recursive call.

We illustrate these proving and synthesis steps by our running example.

Example 8.12. Using the extended $\mathbb{S}$up calculus, we synthesize the program for the specification of Figure 8.1. With the magic formula corresponding to (8.4),

$$\forall v_0, v_\mathsf{s}, z. \Big(\big(\mathsf{half}(v_0) \simeq 0 \wedge (\mathsf{half}(\sigma_w) \simeq \sigma_y \rightarrow \mathsf{half}(v_\mathsf{s}) \simeq \mathsf{s}(\sigma_y))\big) \rightarrow \mathsf{half}(\mathsf{rec}(v_0, v_\mathsf{s}, z)) \simeq z \Big), \quad (8.26)$$

we obtain the following derivation[4]:

1. $\mathsf{half}(y) \not\simeq \sigma \vee \mathsf{ans}(y)$ [preprocessed specification]

[4]we detail the full derivation produced by Vampire in Section 9.2

2. $\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(\sigma_w) \simeq \sigma_y \vee \mathsf{half}(\sigma_x(z)) \simeq z$ [MagInd with (8.26)]
3. $\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(v_\mathsf{s}) \not\simeq \mathsf{s}(\sigma_y) \vee \mathsf{half}(\sigma_x(z)) \simeq z$ [MagInd with (8.26)]
4. $\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(\sigma_w) \simeq \sigma_y \vee \mathsf{ans}(\mathsf{rec}(v_0, v_\mathsf{s}, \sigma))$ [BR 1, 2]
5. $\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(v_\mathsf{s}) \not\simeq \mathsf{s}(\sigma_y) \vee \mathsf{ans}(\mathsf{rec}(v_0, v_\mathsf{s}, \sigma))$ [BR 1, 3]
6. $\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(v_\mathsf{s}) \not\simeq \mathsf{s}(\mathsf{half}(\sigma_w)) \vee \mathsf{ans}(\mathsf{rec}(v_0, v_\mathsf{s}, \sigma))$ [Sup 4, 5]
7. $\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{half}(v_\mathsf{s}) \not\simeq \mathsf{half}(\mathsf{s}(\mathsf{s}(\sigma_w))) \vee \mathsf{ans}(\mathsf{rec}(v_0, v_\mathsf{s}, \sigma))$ [Sup (H3), 6]
8. $\mathsf{half}(v_0) \not\simeq 0 \vee \mathsf{ans}(\mathsf{rec}(v_0, \mathsf{s}(\mathsf{s}(\sigma_w)), \sigma))$ [ER 7]
9. $\mathsf{ans}(\mathsf{rec}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(\sigma_w)), \sigma))$ [BR 8, (H2)]
10. $\square$ [answer literal removal 9]

The program recorded in step 10 of the proof is

$$\mathsf{rec}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(\sigma_w)), x)^\mathsf{R} = \mathsf{R}(\mathsf{s}(0), \lambda\sigma_w.\mathsf{s}(\mathsf{s}(\sigma_w)))(x) = f(x),$$

where f is defined as:

$$f(0) \simeq \mathsf{s}(0)$$
$$f(\mathsf{s}(n)) \simeq \mathsf{s}(\mathsf{s}(f(n)))$$

Note that while the synthesized program satisfies the specification (8.1), it does not match the expected definition of the double function from (8.2). Since the half function is rounding down, and the specification does not require the synthesized function to produce even results, the base case was resolved in step 9 with (H2), leading to $f(0) \simeq \mathsf{s}(0)$. As a result, we have $f(n) = \mathsf{s}(\mathsf{double}(n))$ for any n. $\square$

Example 8.12 demonstrates that specification (8.1) has multiple solutions and saturation can find a solution different from the intended one. In the next example we modify the specification to have a single solution and synthesize it.

Example 8.13. To synthesize the double function, we modify the specification:

$$\text{additional axioms:}\quad \mathsf{even}(0) \tag{E1}$$
$$\neg\mathsf{even}(\mathsf{s}(0)) \tag{E2}$$
$$\forall x.\ (\mathsf{even}(\mathsf{s}(\mathsf{s}(x))) \leftrightarrow \mathsf{even}(x)) \tag{E3}$$
$$\text{new specification:}\quad \forall x \exists y.\ (\mathsf{half}(y) \simeq x \wedge \mathsf{even}(y)) \tag{8.27}$$

After negating and skolemizing (8.27) and adding the answer literal, we obtain:

$$\mathsf{half}(y) \not\simeq \sigma \vee \neg\mathsf{even}(y) \vee \mathsf{ans}(y) \tag{8.28}$$

In this case we use the magic axiom for the conjunction $G[t,x] := \mathsf{half}(x) \simeq t \wedge \mathsf{even}(x)$:

$$\begin{aligned}&\Big(\exists v_0.(\mathsf{half}(v_0) \simeq 0 \wedge \mathsf{even}(v_0)) \wedge \\ &\qquad \forall y.(\exists w.(\mathsf{half}(w) \simeq y \wedge \mathsf{even}(w)) \rightarrow \exists v_{\mathsf{s}}.(\mathsf{half}(v_{\mathsf{s}}) \simeq \mathsf{s}(y) \wedge \mathsf{even}(v_{\mathsf{s}})))\Big) \\ &\quad \rightarrow \forall z.\exists x.(\mathsf{half}(x) \simeq z \wedge \mathsf{even}(x))\end{aligned} \tag{8.29}$$

We clausify the magic formula corresponding to (8.29), and further resolve it with the premise (8.28) to obtain:

$$\begin{gathered}\mathsf{half}(v_0) \not\simeq 0 \vee \neg\mathsf{even}(v_0) \vee \mathsf{half}(\sigma_w) \simeq \sigma_y \vee \mathsf{ans}(\mathsf{rec}(v_0, v_{\mathsf{s}}, \sigma)) \\ \mathsf{half}(v_0) \not\simeq 0 \vee \neg\mathsf{even}(v_0) \vee \mathsf{even}(\sigma_w) \vee \mathsf{ans}(\mathsf{rec}(v_0, v_{\mathsf{s}}, \sigma)) \\ \mathsf{half}(v_0) \not\simeq 0 \vee \neg\mathsf{even}(v_0) \vee \mathsf{half}(v_{\mathsf{s}}) \not\simeq \mathsf{s}(\sigma_y) \vee \neg\mathsf{even}(v_{\mathsf{s}}) \vee \mathsf{ans}(\mathsf{rec}(v_0, v_{\mathsf{s}}, \sigma))\end{gathered}$$

The refutation of these clauses follows a similar course to the proof in Example 8.12. However, v_0 occurring in the literal $\neg\mathsf{even}(v_0)$ forces the proof to use (H1) instead of (H2), and thus the final derived answer literal will be $\mathsf{rec}(0, \mathsf{s}(\mathsf{s}(\sigma_w)), \sigma)$, corresponding exactly to the function definition of double from (8.2). Note that a derivation of this program in this case requires a saturation prover to apply induction on conjunctions of literals. □

8.6 Generalization to Arbitrary Term Algebras

Our approach from Sections 8.3–8.5 generalizes naturally to arbitrary term algebras. In this section we present all definitions, lemmas, and theorems for this generalization.

We will work with an arbitrary (possibly polymorphic) term algebra τ with constructors $\{c_1, \dots, c_n\}$, where we denote the sort of each c_i by $\tau_{i,1} \times \dots \times \tau_{i,n_{c_i}} \rightarrow \tau$, and $P_{c_i} = \{j_1, \dots, j_{|P_{c_i}|}\}$ for each $i = 1, \dots, n$. Let α be any sort. We note that in this section, we consider variables of the sorts given as follows: $x, v, w : \alpha$, and $y, z : \tau$, all possibly with indices.

The *magic axiom for* $G[t,x]$, where $t : \tau, x : \alpha$, and by $\overline{y_c}$ we denote $y_{c,1}, \dots, y_{c,n_c}$, is:

$$\Big(\bigwedge_{c \in \Sigma_\tau} \forall_{i=1}^{n_c} y_{c,i}.((\bigwedge_{j \in P_c} \exists w_{c,j}.G[y_{c,j}, w_{c,j}]) \rightarrow \exists v_c.G[c(\overline{y_c}), v_c])\Big) \rightarrow \forall z.\exists x.G[z,x] \tag{8.30}$$

We use the magic axiom in MagInd, when $L[t,x]$ is a literal with the only free variable x:

$$\frac{\overline{L}[t,x] \vee C}{\Big(\bigwedge_{c \in \Sigma_\tau} \forall_{i=1}^{n_c} y_{c,i}.((\bigwedge_{j \in P_c} \exists w_{c,j}.L[y_{c,j}, w_{c,j}]) \rightarrow \exists v_c.L[c(\overline{y_c}), v_c])\Big) \rightarrow \forall z.\exists x.L[z,x]}\ (\mathsf{MagInd})$$

We convert (8.30) to prenex normal form such that $\forall_{c \in \Sigma_\tau} v_c$ precedes $\exists x$:

$$\begin{aligned}&\exists_{c \in \Sigma_\tau, i \in \{1,\dots,n_c\}} y_{c,i}.\exists_{c \in \Sigma_\tau, k \in P_c} w_{c,k}.\forall_{c \in \Sigma_\tau} v_c.\forall z.\exists x. \\ &\qquad \Big(\bigwedge_{c \in \Sigma_\tau} ((\bigwedge_{j \in P_c} G[y_{c,j}, w_{c,j}]) \rightarrow G[c(\overline{y_c}), v_c]) \rightarrow G[z,x]\Big)\end{aligned} \tag{8.31}$$

We define the *primitive recursion operator* R *for* τ *and* α analogously to Definition 8.2:

$$\begin{aligned}\mathsf{R}(f_1,\dots,f_n)(c_1(\overline{x})) &\simeq f_1(x_1,\dots,x_{n_{c_1}},\mathsf{R}(f_1,\dots,f_n)(x_{j_1}),\dots,\mathsf{R}(f_1,\dots,f_n)(x_{j_{|P_{c_1}|}}))\\ &\vdots\\ \mathsf{R}(f_1,\dots,f_n)(c_n(\overline{x})) &\simeq f_n(x_1,\dots,x_{n_{c_n}},\mathsf{R}(f_1,\dots,f_n)(x_{j_1}),\dots,\mathsf{R}(f_1,\dots,f_n)(x_{j_{|P_{c_n}|}}))\end{aligned}$$

where for each i we have $f_i : \tau_{i,1}\times\dots\times\tau_{i,n_{c_i}}\times\alpha^{|P_{c_i}|}\to\alpha$.

Using the recursion operator R, we state the analogue of Lemma 8.3:

Lemma 8.14 (Recursive Witness for Term Algebra τ)**.** The expression

$$\mathsf{R}(\lambda_{i=1}^{n_{c_1}} y_{c_1,i}.\lambda_{k\in P_{c_1}} w_{c_1,k}.\ v_{c_1},\ \dots,\ \lambda_{i=1}^{n_{c_n}} y_{c_n,i}.\lambda_{k\in P_{c_n}} w_{c_n,k}.\ v_{c_n})(z) \tag{8.32}$$

is a witness for the variable x in axiom (8.31).

Proof. The proof is analogous to the proof of Lemma 8.3. We consider an interpretation I under which (8.32) is not a witness for x in (8.31), and extend it to $J_{\overline{a},\overline{b}}$, parametrized by values $\overline{a},\overline{b}$ assigned to $\overline{y},\overline{w}$. Under this interpretation the antecedent of (8.31) is true. Hence, we obtain one assumption per each of the cases of the antecedent (corresponding to constructors of τ), similarly as we obtained (8.12) for $\mathsf{0}$ and (8.16) for s in the proof of Lemma 8.3. We use these assumptions to refute that there is a smallest value v_z for which

$$G^I[v_z,\mathsf{R}^I(\overline{f})(v_z)]=\bot,$$

where $v_z = z^{J_{\overline{a},\overline{b}}}$ and each element of $\overline{f}$ is defined analogously to f in the original proof. $\square$

For each $G[t,x]$ we introduce a distinct computable function symbol $\mathsf{rec}_{G[t,x]}:\alpha^{n_c}\times\tau\to\alpha$. As for natural numbers, we call such symbols for any $G[t,x]$ the rec-*symbols*, and terms with a rec-symbol as the top-level functor the rec-*terms*.

The *magic formula for* $G[t,x]$ corresponding to magic axiom (8.30) is

$$\forall_{c\in\Sigma_\tau} v_c.\forall z.\Big(\bigwedge_{c\in\Sigma_\tau}\big(\bigwedge_{j\in P_c} G[\sigma_{y_{c,j}},\sigma_{w_{c,j}}]\to G[c(\overline{\sigma_{y_c}}),v_c]\big)\to G[z,\mathsf{rec}_{G[t,x]}(\overline{v},z)]\Big), \tag{8.33}$$

where skolem constants $\sigma_{y_{c_i,j}},\sigma_{w_{c_i,j}}$ are used to skolemize the variables $y_{c_i,j},w_{c_i,j}$, and the skolem function $\mathsf{rec}_{G[t,x]}$ to skolemize the variable x, and where by $\overline{v}$ we denote $v_{c_1},\dots,v_{c_n}$, and by $\overline{\sigma_{y_c}}$ we denote $\sigma_{y_c,1},\dots,\sigma_{y_c,n_c}$ As for natural numbers, we say that the skolem constants $\sigma_{y_{c_i,j}},\sigma_{w_{c_i,j}}$ introduced in the same (8.33) as the $\mathsf{rec}_{G[t,x]}$-term are *associated with the* $\mathsf{rec}_{G[t,x]}$*-term.* Each $\sigma_{y_{c_i,j}},\sigma_{w_{c_i,j}}$ introduced in (8.33) is considered computable only in the ith argument of its associated rec-term.

Exactly as for natural numbers, an inference system $\mathcal{I}$ is rec-*compliant* if:

1. $\mathcal{I}$ only introduces rec-terms in the instances of the magic formula (8.33),
2. $\mathcal{I}$ does not introduce uncomputable symbols into arguments of rec-terms in clauses it derives.

When $\sigma_{y_{c_i,j}}, \sigma_{w_{c_i,j}}$ are associated with $\mathsf{rec}(\overline{s}, t)$, then the term

$$\mathsf{R}(\lambda_{i=1}^{n_{c_1}}\sigma_{y_{c_1,i}}.\lambda_{k\in P_{c_1}}\sigma_{w_{c_1,k}}.\ s_1,\ \ldots,\ \lambda_{i=1}^{n_{c_n}}\sigma_{y_{c_n,i}}.\lambda_{k\in P_{c_n}}\sigma_{w_{c_n,k}}.\ s_n)(t)$$

is the *recursive function term corresponding to* $\mathsf{rec}(\overline{s}, t)$. As for natural numbers, for a term r, we denote by r^{R} the expression obtained from r by iteratively replacing all rec-terms by their corresponding recursive function terms, starting from the innermost ones. Similarly, formula F^{R} denotes the formula F in which we replace all rec-terms by their corresponding recursive function terms.

Lemma 8.15 (Recursive Witness for Magic Formulas Using τ)**.** Consider the formula obtained from (8.33) by replacing $\mathsf{rec}_{G[t,x]}(\overline{v}, z)$ by its corresponding recursive function term:

$$\begin{aligned}\forall_{c\in\Sigma_\tau} v_c.\forall z.\Big(&\bigwedge_{c\in\Sigma_\tau}(\bigwedge_{j\in P_c} G[\sigma_{y_{c,j}}, \sigma_{w_{c,j}}] \to G[c(\overline{\sigma_{y_c}}), v_c]) \\ &\to G[z, \mathsf{R}(\lambda_{i=1}^{n_{c_1}}\sigma_{y_{c_1,i}}.\lambda_{k\in P_{c_1}}\sigma_{w_{c_1,k}}.\ v_{c_1},\ \ldots,\ \lambda_{i=1}^{n_{c_n}}\sigma_{y_{c_n,i}}.\lambda_{k\in P_{c_n}}\sigma_{w_{c_n,k}}.\ v_{c_n})(z)]\Big)\end{aligned} \tag{8.34}$$

For every interpretation, there exists its extension by some

$$\{\sigma_{y_{c,i}} \mapsto v_{y,c,i}, \sigma_{w_{c,k}} \mapsto v_{w,c,k}\}_{c\in\Sigma_\tau, i\in\{1,\ldots,n_c\}, k\in P_c},$$

such that the extension is a model of (8.34). As a consequence, formula (8.34) is satisfiable.

Proof. Immediately follows from (8.34) being a skolemization of

$$\begin{aligned}\exists_{c\in\Sigma_\tau,\, i\in\{1,\ldots,n_c\}} y_{c,i}.\exists_{c\in\Sigma_\tau,\, k\in P_c} w_{c,k}.\forall_{c\in\Sigma_\tau} v_c.\forall z.\Bigg(&\bigwedge_{c\in\Sigma_\tau}((\bigwedge_{j\in P_c} G[y_{c,j}, w_{c,j}]) \to G[c(\overline{y_c}), v_c]) \\ &\to G[z, \mathsf{R}(\lambda_{i=1}^{n_{c_1}} y_{c_1,i}.\lambda_{k\in P_{c_1}} w_{c_1,k}.v_{c_1}, \ldots, \lambda_{i=1}^{n_{c_n}} y_{c_n,i}.\lambda_{k\in P_{c_n}} w_{c_n,k}.v_{c_n})(z)]\Bigg),\end{aligned}$$

which is by Lemma 8.14 valid. □

Using Lemma 8.15, we derive the analogues of Theorems 8.9–8.11 for an arbitrary term algebra τ. Note that since we extended the definition of a rec-compliant system and $r^{\mathsf{R}}, F^{\mathsf{R}}$ for τ, the statements and proofs of the theorems do not change.

We finally note that our synthesis method generalizes also to other sorts as term algebras, as long as the induction axiom used for the sort carries the constructive meaning described in Section 8.2.

CHAPTER 9

Synthesis Examples and Implementation in Vampire

The contributions of this chapter are based on:

Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov. Program Synthesis in Saturation. In Brigitte Pientka and Cesare Tinelli, editors, Proc. of CADE, *volume 14132 of* LNCS, *pages 307–324, Cham, 2023. Springer [HKNV23],*

Petra Hozzová, Daneshvar Amrollahi, Márton Hajdu, Laura Kovács, Andrei Voronkov, and Eva Maria Wagner. Synthesis of Recursive Programs in Saturation. EasyChair Preprint no. 12145, EasyChair, 2024 [HAH$^+$24], to appear in Proc. of IJCAR *2024*

In this chapter we describe the implementation of the methods from chapters 7 and 8. We also present a set of examples for recursion-free as well as recursive program synthesis and experimentally evaluate our methods on them.

We implemented our methods in the first-order theorem prover Vampire [KV13], and the implementation is available online at

`https://github.com/vprover/vampire,`

for recursion-free synthesis in the `master` branch, for synthesis of recursive programs in the `synthesis-recursive` branch.

Our benchmarks as well as the configurations for our experiments are available at:

`https://github.com/vprover/vampire_benchmarks/tree/master/synthesis`

9.1 Recursion-Free Synthesis

We implemented the method from Chapter 7 in Vampire and compared it to the SyGuS-based synthesizer of cvc5 [BBB+22], demonstrating that our approach complements the existing work.

9.1.1 Implementation

We implemented Algorithm 7.1, the modification of Avatar described in Section 7.6, and selected rules of the modified superposition calculus from Figure 7.2.[1] Our implementation, consisting of approximately 1100 lines of C++ code, is based on Vampire's existing support for answer literals for question answering [Reg18], which we extended to synthesis. The synthesis functionality can be turned on using the option `--question_answering synthesis`.

Vampire accepts functional specifications in an extension of the SMT-LIB 2.6 format [BFT16], by using the new command `assert-not` to mark the specification. We consider interpreted theory symbols to be computable. Uninterpreted symbols can be annotated as uncomputable via the command:

```
(set-option :uncomputable (symbol1 ...  symbolN))
```

Vampire also allows specifications in the TPTP format [Sut22] using the `conjecture` or `negated_conjecture` formula roles, but without uncomputable annotations.

Our implementation simplifies the programs we synthesize. First, as mentioned in Section 7.3, we only use a program with conditions recorded during the proof search if the condition appears in the derivation of $\square$. If in Algorithm 7.1 we record a program $\langle z, F\rangle$ where z is a variable, we do not use this program in the final program construction (line 12 of Algorithm 7.1) even if F occurs in the derivation of $\square$ (see Example 7.9). This is because z represents an arbitrary witness – therefore any other program $\langle t, F'\rangle$ (where t is not a variable) computes a satisfactory witness also for the condition F. If all recorded programs correspond to a variable, then we choose a constant of the corresponding sort, and use that as the final program. Further, we utilize the simplification rules of the theorem prover. In the saturation loop of Algorithm 7.1 we first simplify clause C_i, and only then carry out the program recording and answer literal removal on lines 6-10.

9.1.2 Experiments

The goal of our experimental evaluation is to showcase the benefits of our approach on problems that are deemed to be hard, even unsolvable, by state-of-the-art synthesis techniques. We therefore focused on first-order theory reasoning and evaluated our work

[1] Note that Vampire does not yet use the unification with abstraction from Section 7.5. All our examples – including Example 7.10 – can be solved by guiding the proof search such that a rule with ordinary unifier θ is used only if $(\theta, \square)$ is a computable unifier.

on the group theory problems of Examples 7.1, 7.9, and 7.10, as well as on integer arithmetic problems.

As the SMT-LIB format can easily be translated into the SyGuS 2.1 syntax [PPR+21], we compared our results to CVC5 1.0.4 [BBB+22], supporting SyGuS-based synthesis [AFP+19]. Our experiments were run on an AMD Epyc 7502, 2.5 GHz CPU with 1 TB RAM, using a 5-minute time limit per example.

Experimental results with group theory properties. VAMPIRE synthesizes the solutions of the Examples 7.1, 7.9, and 7.10 in 0.01, 13, and 0.03 seconds, respectively. Since these examples use uninterpreted functions, they cannot be encoded in the SyGuS 2.1 syntax, showcasing the limits of other synthesis tools.

Experimental results with maximum of $n \geq 2$ integers. We consider formula (7.13), specifying the maximum of n integers. For $n = 2$, the specification is

$$\forall x_1, x_2 \in \mathbb{Z}.\ \exists y \in \mathbb{Z}.\ \big(y \geq_{\mathbb{Z}} x_1 \wedge y \geq_{\mathbb{Z}} x_2 \wedge (y \simeq x_1 \vee y \simeq x_2)\big),$$

and the program we synthesize is `if` $x_1 <_{\mathbb{Z}} x_2$ `then` x_2 `else` x_1. Thanks to the integration with AVATAR, VAMPIRE (using Z3 [DMB08] as the SMT solver backend) is able to synthesize programs choosing the maximal value for up to $n = 23$ input variables within a 5-minute time limit, exactly as CVC5. For $n > 23$, both VAMPIRE and CVC5 time out. The following table shows times in seconds it took VAMPIRE and CVC5 to solve different versions of the benchmark:

Number n of variables for which max is synthesized	2	5	10	15	20	22	23
VAMPIRE	0.03	0.03	0.05	1	13	55	215
CVC5	0.01	0.03	0.6	6.8	88	188	257

Experimental results with polynomial equations. VAMPIRE can synthesize the solution of polynomial equations; for example, for

$$\forall x_1, x_2 \in \mathbb{Z}.\ \exists y \in \mathbb{Z}.\ y^2 \simeq x_1^2 +_{\mathbb{Z}} 2x_1x_2 +_{\mathbb{Z}} x_2^2,$$

we synthesize $x_1 +_{\mathbb{Z}} x_2$. VAMPIRE finds the corresponding program in 26 seconds using simple first-order reasoning, while CVC5 fails in our setup, even with the help of complex decision procedures for non-linear arithmetic.

9.2 Synthesis of Recursive Programs

We instrumented VAMPIRE with a proof-of-concept implementation of our method from Chapter 8. We also present a collection of problems for which our framework synthesizes programs.

Specification	Program	Synthesized definitions	Vampire
Double: $\forall x \in \mathbb{N}.\exists y \in \mathbb{N}.$ $(\mathsf{half}(y) \simeq x \land \mathsf{even}(y))$	$f(x)$	$f(\mathsf{0}) \simeq \mathsf{0}$ $f(\mathsf{s}(n)) \simeq \mathsf{s}(\mathsf{s}(f(n)))$	✓
Associativity of addition: $\forall x_1, x_2, x_3 \in \mathbb{N}.\exists y \in \mathbb{N}.$ $(x_1 +_{\mathbb{N}} x_2) +_{\mathbb{N}} x_3 \simeq x_1 +_{\mathbb{N}} y$	$f(x_3)$	$f(\mathsf{0}) \simeq x_2$ $f(\mathsf{s}(n)) \simeq \mathsf{s}(f(n))$	✓
Subtraction with condition: $\forall x_1, x_2 \in \mathbb{N}.\exists y \in \mathbb{N}.$ $(x_2 <_{\mathbb{N}} x_1 \rightarrow x_2 +_{\mathbb{N}} y \simeq x_1)$	$f(x_2)$	$f(\mathsf{0}) \simeq x_1$ $f(\mathsf{s}(n)) \simeq \mathsf{p}(f(n))$	✓
Floored square root: $\forall x \in \mathbb{N}.\exists y \in \mathbb{N}.$ $(y \cdot_{\mathbb{N}} y \leq_{\mathbb{N}} x \land x <_{\mathbb{N}} \mathsf{s}(y) \cdot_{\mathbb{N}} \mathsf{s}(y))$	$f(x)$	$f(\mathsf{0}) \simeq \mathsf{0}$ $f(\mathsf{s}(n)) \simeq \mathtt{if}\ \mathsf{s}(n) \simeq \mathsf{s}(f(n)) \cdot_{\mathbb{N}} \mathsf{s}(f(n))$ $\mathtt{then}\ \mathsf{s}(f(n))\ \mathtt{else}\ f(n)$	–
Floored division: $\forall x_1, x_2 \in \mathbb{N}.\exists y \in \mathbb{N}.(x_2 \not\simeq 0 \rightarrow$ $(y \cdot_{\mathbb{N}} x_2 \leq_{\mathbb{N}} x_1 \land x_1 <_{\mathbb{N}} \mathsf{s}(y) \cdot_{\mathbb{N}} x_2))$	$f(x_1)$	$f(\mathsf{0}) \simeq \mathsf{0}$ $f(\mathsf{s}(n)) \simeq \mathtt{if}\ \mathsf{s}(n) \simeq \mathsf{s}(f(n)) \cdot_{\mathbb{N}} x_2$ $\mathtt{then}\ \mathsf{s}(f(n))\ \mathtt{else}\ f(n)$	–
Length of two concatenated lists: $\forall x_1, x_2 \in \mathbb{L}.\exists y \in \mathbb{N}.$ $y \simeq \mathsf{len}(x_1 \mathbin{+\!\!+} x_2)$	$f(x_1)$	$f(\mathsf{nil}) \simeq \mathsf{len}(x_2)$ $f(\mathsf{cons}(n,l)) \simeq \mathsf{s}(f(l))$	✓
Last element of a list: $\forall x \in \mathbb{L}.\exists y \in \mathbb{N}.(x \not\simeq \mathsf{nil} \rightarrow$ $\exists z \in \mathbb{L}.x \simeq z \mathbin{+\!\!+} \mathsf{cons}(y, \mathsf{nil}))$	$f(x)$	$f(\mathsf{cons}(n,\mathsf{nil})) \simeq n$ $l \not\simeq \mathsf{nil} \rightarrow f(\mathsf{cons}(n,l)) \simeq f(l)$	✓
Prefix of a list given its suffix: $\forall x_1, x_2 \in \mathbb{L}.\exists y \in \mathbb{L}.$ $(\mathsf{suff}(x_2, x_1) \rightarrow x_1 \simeq y \mathbin{+\!\!+} x_2)$	$f(x_2)$	$f(\mathsf{nil}) \simeq x_1$ $f(\mathsf{cons}(n,l)) \simeq g(f(l))$ $g(\mathsf{cons}(n,\mathsf{nil})) \simeq \mathsf{nil}$ $l \not\simeq \mathsf{nil} \rightarrow g(\mathsf{cons}(n,l)) \simeq \mathsf{cons}(n,g(l))$	–
Maximum element of a list: $\forall x \in \mathbb{L}.\exists y \in \mathbb{N}.(x \not\simeq \mathsf{nil} \rightarrow$ $(\mathsf{in}_{\mathbb{N}}(y,x) \land \forall k \in \mathbb{N}.(\mathsf{in}_{\mathbb{N}}(k,x) \rightarrow k \leq_{\mathbb{N}} y))$	$f(x)$	$f(\mathsf{cons}(n,\mathsf{nil})) \simeq n$ $l \not\simeq \mathsf{nil} \rightarrow f(\mathsf{cons}(n,l)) \simeq \mathtt{if}\ f(l) <_{\mathbb{N}} n$ $\mathtt{then}\ n\ \mathtt{else}\ f(l)$	–
Maximum element of a tree: $\forall x \in \mathbb{BT}.\exists y \in \mathbb{N}.(x \not\simeq \mathsf{Nil} \rightarrow$ $(\mathsf{in}_{\mathbb{BT}}(y,x) \land \forall k \in \mathbb{N}.(\mathsf{in}_{\mathbb{BT}}(k,x) \rightarrow k \leq_{\mathbb{N}} y)$	$f(x)$	$f(\mathsf{node}(\mathsf{Nil},n,\mathsf{Nil})) \simeq n$ $r \not\simeq \mathsf{Nil} \rightarrow f(\mathsf{node}(\mathsf{Nil},n,r)) \simeq$ $\mathtt{if}\ n <_{\mathbb{N}} f(r)\ \mathtt{then}\ f(r)\ \mathtt{else}\ n$ $l \not\simeq \mathsf{Nil} \rightarrow f(\mathsf{node}(l,n,\mathsf{Nil})) \simeq$ $\mathtt{if}\ n <_{\mathbb{N}} f(l)\ \mathtt{then}\ f(l)\ \mathtt{else}\ n$ $l \not\simeq \mathsf{Nil} \land r \not\simeq \mathsf{Nil} \rightarrow f(\mathsf{node}(l,n,r)) \simeq$ $\mathtt{if}\ f(l) <_{\mathbb{N}} f(r)\ \mathtt{then}$ $\mathtt{if}\ n <_{\mathbb{N}} f(r)\ \mathtt{then}\ f(r)\ \mathtt{else}\ n$ $\mathtt{else}$ $\mathtt{if}\ n <_{\mathbb{N}} f(l)\ \mathtt{then}\ f(l)\ \mathtt{else}\ n$	–

Table 9.1: Synthesis examples using natural numbers $\mathbb{N}$, lists $\mathbb{L}$ and binary trees $\mathbb{BT}$. The x-variables in the program and synthesized definitions are the inputs. While our framework synthesizes all these examples, our implementation in Vampire only synthesizes those marked with "✓". Note that for "Length of 2 concatenated lists" we consider $\mathbin{+\!\!+}$ to be uncomputable.

9.2.1 Implementation

We implemented our method for recursive program synthesis in saturation. Our implementation consists of approximately 1,100 lines of C++ code on top of the recursion-free synthesis implementation from the previous section. As a proof of concept, it supports the MagInd rule with structural induction axiom for general term algebras. Additionally, we also implemented a version of MagInd using a magic axiom with base case $\mathsf{s}(0)$ for natural numbers and $\mathsf{cons}(a, \mathsf{nil})$ for any a for lists.

To support synthesis requiring induction on specifications $\neg F[t, x]$, where $F[t, x]$ is an arbitrary formula with the only free variable x, we use an encoding as follows. We change the original specification $\forall \overline{x}.\exists y.F[\overline{x}, y]$ to $\forall \overline{x}.\exists y.p(\overline{x}, y)$, where p is a fresh uncomputable predicate, and we add an axiom $\forall \overline{x}, y.(p(\overline{x}, y) \leftrightarrow F[\overline{x}, y])$. This encoding allows us to synthesize more problems in practice. However, it is not a universal replacement of a rule which allows induction on complex formulas, because we can only use it if the formula, on which induction should be applied, occurs already in the input.

9.2.2 Examples

Our implementation can synthesize the programs for the specification (8.1), and using the encoding mentioned above, also the program for (8.27), both in under 1 second. We also synthesize further examples over natural numbers $\mathbb{N}$, lists $\mathbb{L}$, and binary trees $\mathbb{BT}$. We display the specifications alongside the programs synthesized by our framework in Table 9.1. For the full derivations of most of the synthesized programs, see Appendix D of [HAH+24]. Our framework synthesizes programs for each of the examples, yet our implementation supports so far only a limited set of magic formulas; therefore, the "VAMPIRE" column of Table 9.1 lists which examples are solved in practice.

Note that for the second example of Table 9.1 (associativity of addition), a possible program would be $x_2 +_{\mathbb{N}} x_3$. Using our framework we however synthesize a *syntactically* different program. The function $f(n)$ we synthesize for this example computes $x_2 +_{\mathbb{N}} n$, and the program is $f(x_3)$, which is *semantically* equivalent to $x_2 +_{\mathbb{N}} x_3$. Further, for the sixth example (length of two concatenated lists), we consider $+\!\!+$ to be uncomputable to disallow the trivial derivation consisting of one application of equality resolution, deriving the term from specification $\mathsf{len}(x_1 +\!\!+ x_2)$ as the program. For the eighth example (prefix of a list given suffix) we construct a recursive function calling another recursive function. Intuitively, $g(l)$ removes the last element of l, while $f(l')$ iteratively calls g on x_1 as many times as there are elements in l'. The synthesized program for this example is $f(x_2)$, resulting in removing $\mathsf{len}(x_2)$ elements from the end of x_1. Finally, note that to derive the programs for the last four examples, we use induction axioms with non-nil and non-Nil base cases.

Vampire derivation. Finally, we list the derivation produced by VAMPIRE for specification (7.4), corresponding to Example 8.12. The runtime was 0.021s. The derivation uses the TPTP syntax [Sut22]. We underline the final derived program consisting of the definition of the function `rf85` (corresponds to f from Example 8.12), and the program body `rf85(X0)`, where the variable `X0` to the input variable x.

Note that the derivations produced by Vampire might differ from those presented in Chapter 8 due to Vampire using specific ordering and selection constraints, and a limited subset of MagInd instances.

Vampire configuration used to produce this derivation:
`--forced_options ind=struct:indu=off:qa=synthesis`

Output:

```
...
% Inputs for synthesis:
5. ~! [X0 : nat] : ? [X1 : nat] : half(X1) = X0 [negated conjecture
   4]
% Recursive function definitions:
rf85(zero) = s(zero)
rf85(s(X5)) = s(s(rf85(X5)))
% SZS answers Tuple [[rf85(X0)]|_]
% SZS output start Proof
2. zero = half(s(zero)) [input]
3. ! [X0 : nat] : s(half(X0)) = half(s(s(X0))) [input]
4. ! [X0 : nat] : ? [X1 : nat] : half(X1) = X0 [input]
5. ~! [X0 : nat] : ? [X1 : nat] : half(X1) = X0 [negated conjecture
   4]
9. ! [X1 : nat] : ~(half(X1) = sK1_in & ans0(X1)) [answer literal
   with input var skolemisation 5]
10. ! [X0 : nat] : ~(half(X0) = sK1_in & ans0(X0)) [rectify 9]
11. ! [X0 : nat] : (half(X0) != sK1_in | ~ans0(X0)) [ennf
   transformation 10]
12. half(X0) != sK1_in | ~ans0(X0) [cnf transformation 11]
13. s(half(X0)) = half(s(s(X0))) [cnf transformation 3]
14. zero = half(s(zero)) [cnf transformation 2]
20. ? [X5 : nat] : ? [X6 : nat] : ! [X7 : nat,X3 : nat] : ! [X8 : nat
   ] : ((zero = half(X3) & (half(X6) = X5 => s(X5) = half(X7))) =>
   half(rec2(X3,X7,X8)) = X8) [structural induction hypothesis]
21. ? [X5 : nat] : ? [X6 : nat] : ! [X7 : nat,X3 : nat] : ! [X8 : nat
   ] : (half(rec2(X3,X7,X8)) = X8 | (zero != half(X3) | (s(X5) !=
   half(X7) & half(X6) = X5))) [ennf transformation 20]
22. sK3 = half(sK4) | zero != half(X3) | half(rec2(X3,X7,X8)) = X8 [
   cnf transformation 21]
23. half(X7) != s(sK3) | zero != half(X3) | half(rec2(X3,X7,X8)) = X8
    [cnf transformation 21]
24. half(X1) != s(sK3) | zero != half(X0) | ~ans0(rec2(X0,X1,sK1_in))
    [resolution 23,12]
25. zero != half(X0) | sK3 = half(sK4) | ~ans0(rec2(X0,X1,sK1_in)) [
   resolution 22,12]
123. zero != zero | sK3 = half(sK4) | ~ans0(rec2(s(zero),X0,sK1_in))
   [superposition 25,14]
127. sK3 = half(sK4) | ~ans0(rec2(s(zero),X0,sK1_in)) [trivial
   inequality removal 123]
160. s(half(X0)) != s(sK3) | zero != half(X1) | ~ans0(rec2(X1,s(s(X0)
   ),sK1_in)) [superposition 24,13]
724. s(sK3) != s(sK3) | zero != half(s(zero)) | ~ans0(rec2(s(zero),s(
   s(sK4)),sK1_in)) [superposition 160,127]
753. zero != half(s(zero)) | ~ans0(rec2(s(zero),s(s(sK4)),sK1_in)) [
   trivial inequality removal 724]
758. ~ans0(rec2(s(zero),s(s(sK4)),sK1_in)) [subsumption resolution
   753,14]
759. ans0(X0) [answer literal]
760. $false [unit resulting resolution 759,758]
% SZS output end Proof
% ------------------------------
% Version: Vampire 4.8 (commit 3cddf8311 on 2024-01-28 09:37:47
   +0100)
% Termination reason: Refutation
% Memory used [KB]: 718
% Time elapsed: 0.021 s
% ------------------------------
```

CHAPTER 10

Related Work

In this chapter we outline related work and discuss its relationship with our contributions.

10.1 Induction

Research in automating induction has a long history with a number of techniques developed, including for example approaches based on semi-automatic inductive theorem proving [BM79, BSvH$^+$93, PCI$^+$20, CJRS12], specialized rewriting procedures [FK12], SMT reasoning [RK15] and superposition reasoning [KP13, Cru17, RV19, EP20].

Previous works on automating induction mainly focus on inductive theorem proving [BSvH$^+$93, CJRS12, SDE12]: deciding when induction should be applied and what induction axiom should be used. Further restrictions are made on the logical expressiveness, for example, induction only over universal properties [BM79, SDE12] and without uninterpreted symbols [PCI$^+$20], or only over term algebras [KP13, EP20]. Inductive provers usually rely on auxiliary lemmas to help prove an inductive property. In [CJRS12] heuristics for finding such lemmas are introduced, for example by randomly generating equational formulas over random inputs and using these formulas if they hold reasonably often. The use of [CJRS12] is therefore limited to the underlying heuristics. Other approaches to automating induction circumvent the need for auxiliary lemmas by using uncommon cut-free proof systems for inductive reasoning, such as a restricted ω-rule [BIS92], or cyclic reasoning [BS11].

Our work from Chapters 3 and 4, extending [RV19], automates induction by integrating it directly in superposition-based proof search, without relying on rewrite rules and external heuristics for generating auxiliary inductive lemmas/subgoals as in [BSvH$^+$93, CJRS12, SDE12, BM79, PCI$^+$20]. The work was also later extended in [HHKV21] and [HKRV22] to support induction based on recursive function definitions and induction on complex formulas. See also [HHK$^+$22] for a survey of induction methods used by Vampire.

10.1.1 Induction with Generalization

Our new inference rule IndGen for induction with generalization (Chapter 3) adds new formulas to the search space and can thus in some cases play the role of lemma discovery heuristics used in [BSvH+93, CJRS12, RK15]. Our work also extends [RV19] by using and instantiating induction axioms with logically stronger versions of the property being proved. Unlike [Cru17], our methods do not necessarily depend on Avatar [Vor14], and can be used with any sort (not just inductive data types) and target also induction rules different than structural induction. Contrarily to [EP20], we are not limited to induction over term algebras with the subterm ordering and we stay in the standard saturation framework. Moreover, compared to [BM79, BSvH+93, CJRS12, RK15, SDE12, PCI+20], one of the main advantages of our approach is that it does not use a goal-subgoal architecture and can, as a result, combine superposition-based equational reasoning with inductive reasoning.

Normally, generalization in theorem proving means that given a goal F, we try to prove a more general goal. In logic, a statement F' is more general than F if F' implies F. Thus, by proving F' we also prove F. One way to generalize is to replace one or more occurrences of a subterm with a fresh variable, using the fact that $\forall x.F[x]$ implies $F[t]$. This is essentially the idea behind approaches to generalization in all systems we compared our method with. While our approach is superficially similar, it does something *fundamentally different*. Instead of (or beside) adding an instance I of the induction schema that can be used to prove $F[t]$, we add an instance I' that can be used to prove $\forall x.F[x]$. An interesting observation is that, in general, neither I implies I', nor I' implies I, so neither of I and I' is more general.

The *second fundamental difference* is that because induction in Vampire is not based on a goal-subgoal architecture, we can add both induction formulas I and I' at the same time. While this may seem inefficient, for some induction schemas, including structural induction, the overhead in practice is negligible (as also confirmed by our experiments).

10.1.2 Integer Induction

Previous works on automating induction mainly focused on inductive reasoning for inductively defined data types, for example in inductive theorem provers ACL2 [KMM00], IsaPlanner [DF04], HipSpec [CJRS13], Zeno [SDE12] and Imandra [PCI+20]; superposition theorem provers Zipperposition [Cru17] and Vampire [RV19]; and the SMT solver CVC4 [RK15] (succeeded by cvc5 [BBB+22]). While most of these solvers support reasoning with integers, only ACL2 and CVC4/cvc5 implement some form of induction over integers.

The ACL2 approach [KMM00] generates induction schemas based on recursive function calls in the property to be proved. Hence, it can only use induction to solve problems based on recursively defined functions, and moreover, only on functions that can be proven terminating. Further, as mentioned in Section 6.2, ACL2 only supports interpreted constants as base cases. As an inductive theorem prover, ACL2 has limitations with respect to theory reasoning and reasoning with quantifiers, which further curb its utility when it comes to program verification and reasoning about integers in general. On the other hand, the SMT-based setting of CVC4 and cvc5 [RK15] has a robust handling of theories, and to an extent also supports reasoning with quantifiers. It applies induction by inductive strengthening of SMT properties in combination with subgoal discovery. However, as noted in Section 6.2, CVC4/cvc5 is limited to upward induction with interpreted constants as base cases.

While downward integer induction can be considered a straightforward extension of upward integer induction and does not solve many more problems in our benchmark sets, symbolic bounds provide a very powerful generalization, as witnessed by our experimental results (see Section 6.2). In automated reasoning, the power provided by more general rules comes with the price of uncontrollable blowup of the search space. To harness this power we came up with defining (interval) upward/downward induction rules with symbolic bounds in the superposition calculus in such a way that they result in most cases in the addition of very simple and ground clauses (see e.g. Example 4.5), which can be efficiently handled within the Avatar architecture.

We believe that variants of our induction rules defined in Section 4.3 can also be successfully used by SMT solvers. The idea is to apply them, like we do, only when there is a suitable bound in the current candidate model. One can also combine this with the observation made in Example 4.5: one can resolve added induction formulas against literals already occurring in the search space to add only ground formulas.

10.1.3 Inductive Benchmarks

The benchmark suite we propose and use in this thesis is new and can be used to complement existing inductive benchmarks: the TIP library [CJRS15] and the examples of [RK15].

Both TIP and the benchmark set of [RK15] focus on classic inductive problems inspired by program verification and mathematical properties. An overwhelming majority of the benchmarks in TIP focus on inductive data types – in fact, out of more than 536 inductive problems, only 3 use integers and no inductive data types. The examples from [RK15] contain 311 inductive benchmarks translated into three encodings: (i) using only inductive data types, (ii) using integers instead of natural numbers, but also other inductive data types (such as lists or trees), and (iii) using both integers and natural numbers to express the same properties, alongside other inductive data types. Problems from (iii) are also included in the UFDTLIA benchmark set of SMT-LIB [BFT16].

While our benchmarks also include 63 classic problems with inductive data types, sharing some of the problems with TIP and [RK15], we also provide a large set of 3,333 problems of increasing sizes for inductive data types (category `dty`). Further, our suite also contains 120 benchmarks targeting different variations of induction over integers (category `int`). Note that there is a substantial difference between our `int` benchmarks and benchmarks from (ii). The latter mostly require inductive reasoning only for inductive data types (or no induction at all): they contain integers but only a few of them require inductive reasoning over integers, while most of our `int` benchmarks require proper integer induction. For example, Vampire can solve 131 of 311 benchmarks in (ii) without using integer induction. In this respect, the available benchmark sets reflect the inductive capabilities of different provers and solvers, which, as mentioned in Subsection 10.1.2, have only limited support for integer induction.

In terms of the benchmark syntax, TIP uses a non-standard variant of SMT-LIB, and offers tools for translating the benchmarks into standard SMT-LIB. All three encodings of [RK15] employ the standard SMT-LIB 2 syntax. Our dataset is also encoded in the current standard SMT-LIB 2 syntax, allowing us to potentially integrate our examples in any repository using the SMT-LIB standard.

10.2 Synthesis

Our work builds upon deductive synthesis [MW80] using answer literals [Gre69] adapted for the resolution calculus [LWC74, Tam95].

The deductive framework of [MW80] combines theorem proving and program synthesis. The framework uses unification, induction, and transformation rules. The synthesized programs can contain recursive functions constructed based on the induction subgoals. Work of [LWC74] adapts this approach for resolution calculus. Further, [Tam95] extends it by an optional restriction on which synthesized programs are allowed (see Section 2.4), and proves completeness of the calculus. The synthesized programs can contain recursion if the proofs use induction outside of the calculus. However, these approaches offer no strategy for proof search, and consequently have no implementation.

We extend this line of work by modifying the superposition calculus and integrating it into the saturation algorithm. We thus reason not only about answer literals but also about their use of **if**−**then**−**else** terms, which we construct not only within answer literals, but also by removal of the answer literals from the clauses in saturation. The modifications of superposition and saturation allow us to construct programs while keeping the impact to the practical proof search efficiency low (see Sections 7.3–7.4).

Further, [MW80, Tam95] construct recursive programs from proofs by induction by reducing the program specification to subgoals corresponding to the cases of the induction axiom. Induction is thus applied outside of their calculus. Modern first-order theorem provers mostly implement saturation-based proof search, which however does not support a goal-subgoal architecture. Our approach integrates induction directly into saturation and thus enables automated inductive reasoning, resulting in automated synthesis of recursive programs.

Recursion-free synthesis. We further overview less closely related work on synthesis, starting with recursion-free synthesis. Component-based synthesis of recursion-free programs [SWL$^+$94] from logical specifications is addressed in [SWL$^+$94, GJTV11, TGD15].

Amphion [SWL+94] uses the first-order theorem prover Snark to prove graphical specifications based on axiomatized subroutine library. The system then extracts recursion-free Fortran programs from proofs. Works of [GJTV11, TGD15] produce $\exists\forall$-formulas to capture specifications over component properties. In [GJTV11], the existential quantifier captures the locations of the components, and the formula is solved by counterexample-guided iterative synthesis using SMT solving. The method of [TGD15] considers specifications consisting of a size constraint, and functional and non-functional requirements on the program. These are combined into the $\exists\forall$-formula, where the existential quantifier captures the sought term. Similarly as in [GJTV11], the formula is solved by SMT solving. Both these approaches produce straight-line programs – however, as `if−then−else` can be encoded as a base component, this corresponds to recursion-free programs.

The sketching technique [SL09, TB13] synthesizes program assignments to variables. Sketch [SL09] uses a C-like programming language, while Rosette [TB13] is an extension to the Racket programming language with high-level meta-programming primitives. Both rely on an alternative framework to our program synthesis setting. In particular, sketching addresses domains that do not involve input logical formulas as functional specifications, such as example-guided synthesis [TNS+21].

A prominent line of research comes with syntax-guided synthesis (SyGuS) [ABD+15], where functional specifications are complemented with a context-free grammar. This grammar yields program templates to be synthesized via an enumerative search procedure, possibly based on SMT solving. SyGuS solvers include DryadSynth [HQSW20] and the SMT solver cvc5 [BBB+22], which offers a SyGuS mode. As such, the SMT-based synthesis techniques belong to a large class of program synthesis techniques referred to as Oracle-Guided Inductive Synthesis [JS17], where synthesis candidates are evaluated by external programs, providing feedback to refine the search. If the oracle provides feedback in the form of counterexamples to refine the search, the method is referred to as a Counter-Example Guided Inductive Synthesis (CEGIS) method [STB+06, SJB08]. We believe our work on recursion-free synthesis by strengthening first-order reasoning for program synthesis is complementary to SyGuS, as evidenced by Examples 7.1, 7.9, and 7.10.

Recursive synthesis. While SyGuS supports specifications for recursive functions and can encode our examples from Section 9.2.2, SyGuS solvers so far do not support recursive synthesis. Further, the semantics-guided synthesis framework SemGuS [KHDR21] is conceptually very general and as such also supports recursive functions. However, its (to the best of our knowledge) only solvers Messy [KHDR21] and Messy-Enum [DHKR21] rely on input specifications different from the FOL setting we use: Messy synthesizes programs from input-output examples, while Messy-Enum requires a grammar to enumerate candidate programs.

Fully automated methods supporting recursive program synthesis include SynQuid [PKSL16], Leon [KKKS13], Jennisys [LM12], SuSLik [PS19], Cypress [IPP+21], and Burst [MNnB+22]. Except for Burst, all these works decompose goals into subgoals. Our work complements these methods, by turning saturation into a recursive synthesis framework reasoning with first-order logic with theories. As such, our work also differs from Synquid, where term enumeration combined with type checking is used over program specifications within decidable logics. Its specifications are formulas using polymorphic refinement types, that is, types augmented by predicates from a decidable logic. Leon uses recursive schemas corresponding to our recursive operator R, instantiates them by candidate program terms, and checks if they satisfy the specification. Additionally, it employs abduction to infer when a particular program branch works only under certain conditions. It works with specifications expressed in a subset of the programming language Scala. Unlike Leon, we support a complete handling of quantifiers via superposition reasoning. Jennisys uses a verifier to generate input-output examples, which differs from our setting of using inductive formulas as logical specifications. These examples are then extrapolated into a heap-manipulating program. The specification consists of a data structure abstraction and definition, and a behavioral description, expressed in Jennisys' own language. SuSLik introduces a separation logic framework dubbed SSL, which derives programs alongside proofs, using goal-directed backtracking proof search. The specification consists of a pre- and post-condition on the heap state encoded in separation logic. Cypress is an extension of SuSLik by cyclic proofs, deriving a wider range of recursive functions. Burst generates programs by composition from existing ones, using quantifier-free fragments of first-order logic. Contrarily to this, we support full first-order logic and induction, without using subgoal proof strategies.

There is also a body of work on semi-automated recursive synthesis, which we do not overview here. Similarly, we do not discuss synthesis based on input-output example pairs nor templates.

CHAPTER 11

Conclusions and Future Work

This thesis focused on automating induction for reasoning about programs. In the first part, we extended the capabilities of induction in saturation [RV19] by induction with generalization and integer induction (Chapters 3, 4, and 6). We also presented a new inductive benchmark set (Chapter 5). In the second part, we developed the saturation-based proving framework into a (recursive) program synthesis framework (Chapters 7-9).

We summarize the conclusions of the individual parts and outline possible directions for future work below.

Induction. We introduced a new inference rule for induction with generalization in saturation-based reasoning. The rule is based on adding induction axioms for proving generalizations of the goals appearing during proof-search. Our experiments show that we solve many problems that other existing systems and approaches cannot solve. Possible directions for future work include designing heuristics to guide proof search and performing other kinds of generalization and induction. We also note that the related direction of using recursive function definitions for rewriting and constructing induction axioms was already addressed in [HHKV21].

Further, we introduced new rules for automating inductive reasoning with integers. In these rules, we instantiate a (symbolic) bound for the induction term as well as the induction step based on comparison literals occurring in the search space. We showed that these rules can be efficiently implemented in saturation-based theorem proving. Many problems in program analysis and mathematical problems of integers previously unsolvable by any theorem prover can now be solved completely automatically. We believe our results can advance automated program analysis and automation of mathematics, where integers are commonly used. This observation also serves as a pointer to a possible area for future work: tighter integration of theorem proving with program analysis, both by incorporating theorem provers into the verification toolchain, but also by extending theorem proving with custom induction schemas tailored to program verification problems.

Inductive benchmarks. We described our benchmark set for evaluating inductive capabilities of automated reasoners. Although we primarily provide our problems in the standard SMT-LIB syntax, we also translated them to other input formats of state-of-the-art reasoners to facilitate comparison of different approaches to inductive reasoning.

An obvious future work direction is to extend our benchmark set with further examples coming from application domains of security and safety verification, as well as formalization of mathematics. Another task for future work is a possible integration of our dataset with the TIP benchmark set [CJRS15] or with the SMT-LIB repository [BFT16]. One possibility for incorporating our benchmark set into SMT-LIB would be to add a new subset or an annotation for inductive problems in SMT-LIB, since SMT-LIB does not currently distinguish benchmarks focused on induction from those which can be easily solved without induction.

Program synthesis. We extended saturation-based proof search to saturation-based program synthesis. Our first aim was to derive recursion-free programs from specifications expressed as forall-exists formulas in first-order logic, augmented with a computability annotation defining which symbols are (not) allowed in the target programs. To this end we integrated the answer literal technique with saturation and defined calculus requirements that ensure that the calculus derives correct (conditional) programs in the form of computable terms. We also modified the superposition calculus and unification in a corresponding way, and demonstrated that our approach synthesizes computable programs. Our initial experiments show that a first-order theorem prover becomes an efficient program synthesizer.

We then extended our framework to synthesize recursive programs by utilizing the constructive nature of induction axioms. We introduced magic axioms as a tracking mechanism and seamlessly integrated these axioms into saturation. Using our framework with these axioms, we construct correct recursive programs, as also demonstrated by our proof-of-concept implementation.

The synthesis of recursive programs could be explored in more depth in the future, similarly to the study of recursion-free synthesis in [Hoz24]. Beyond that, a possible direction for future work is extending our framework with tailored handling of (more general) magic axioms, and respective superposition inferences. Further, the synthesized programs could be simplified in postprocessing.

Another line of future work is extending the specifications our framework supports. One possibility would be to go in the direction of the SyGuS format [PPR$^+$21], where the specification includes a grammar for the language of the target program. A conceptually simpler yet powerful possibility would be to extend the specification language with an interpreted predicate unifies, requiring that the target program unifies (or conversely, does not unify) with a given term [G. Sutcliffe, personal communication, June 6, 2023].

Finally, another interesting direction is to relate synthesis with proving for higher-order logic. Since our synthesis specifications correspond to a certain class of higher-order formulas, one possibility would be to use synthesis in higher-order proving. The connection with higher-order logic also points to further possible specification extensions, such as going beyond single-invocation properties.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ABD$^+$15] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

[AFP$^+$19] Rajeev Alur, Dana Fisman, Saswat Padhi, Andrew Reynolds, Rishabh Singh, and Abhishek Udupa. SyGuS-Comp 2019. `https://sygus.org/comp/2019/`, 2019.

[BBB$^+$22] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Proc. of TACAS*, pages 415–442, 2022.

[BCD$^+$11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. of CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

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

[BIS92] Siani Baker, Andrew Ireland, and Alan Smaill. On the Use of the Constructive Omega-Rule within Automated Deduction. In *Proc. of LPAR*, pages 214–225, 1992.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in computing*. Academic Press, 1979.

[Bon99] Maria Paola Bonacina. A Taxonomy of Theorem-Proving Strategies. In *Artificial Intelligence Today: Recent Trends and Developments*, pages 43–84. 1999.

[BRSV16] Nikolaj Bjøner, Giles Reger, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In *Proc. of GCAI*, pages 39–52, 2016.

[BS11] James Brotherston and Alex Simpson. Sequent Calculi for Induction and Infinite Descent. *J. Log. Comput.*, 21(6):1177–1216, 2011.

[BSvH$^+$93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A Heuristic for Guiding Inductive Proofs. *Artif. Intell.*, 62(2):185–253, 1993.

[CJRS12] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. HipSpec: Automating Inductive Proofs of Program Properties. In *Proc. of ATx/WInG*, pages 16–25, 2012.

[CJRS13] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating Inductive Proofs using Theory Exploration. In M. P. Bonacina, editor, *Proc. of CADE*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.

[CJRS15] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. TIP: Tons of Inductive Problems. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Proc. of CICM*, volume 9150 of *LNCS*, pages 333–337. Springer, 2015.

[Cru17] Simon Cruanes. Superposition with Structural Induction. In C. Dixon and M. Finger, editors, *Proc. of FRoCoS*, volume 10483 of *LNCS*, pages 172–188. Springer, 2017.

[DF04] Lucas Dixon and Jacques Fleuriot. Higher Order Rippling in IsaPlanner. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Proc. of TPHOLs*, volume 3223 of *LNCS*, pages 83–98. Springer, 2004.

[DHKR21] Loris D'Antoni, Qinheping Hu, Jinwoo Kim, and Thomas Reps. Programmable Program Synthesis. In Alexandra Silva and K. Rustan M. Leino, editors, *Proc. of CAV*, pages 84–109, Cham, 2021. Springer.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Proc. of TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[EP20] Mnacho Echenheim and Nicolas Peltier. Combining Induction and Saturation-Based Theorem Proving. *J. Automated Reasoning*, 64:253–294, 2020.

[FK12] Stephan Falke and Deepak Kapur. Rewriting Induction + Linear Arithmetic = Decision Procedure. In *Proc. of IJCAR*, pages 241–255, 2012.

[FPMG19] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified Invariants via Syntax-Guided Synthesis. In I. Dillig and S. Tasiran, editors, *Proc. of CAV*, volume 11561 of *LNCS*, pages 259–277. Springer, 2019.

[GGK20] Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace Logic for Inductive Loop Reasoning. In Alexander Ivrii and Ofer Strichman, editors, *Proc. of FMCAD*, volume 1 of *Conference Series: FMCAD*, pages 255–263, 2020.

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-Free Programs. In *Proc. of PLDI*, page 62–73, 2011.

[Gre69] Cordell Green. Theorem-Proving by Resolution as a Basis for Question-Answering Systems. *Machine Intelligence*, 4:183–205, 1969.

[HAH$^+$24] Petra Hozzová, Daneshvar Amrollahi, Márton Hajdu, Laura Kovács, Andrei Voronkov, and Eva Maria Wagner. Synthesis of Recursive Programs in Saturation. EasyChair Preprint no. 12145, EasyChair, 2024.

[HBNR23] Petra Hozzová, Jaroslav Bendík, Alexander Nutz, and Yoav Rodeh. Overapproximation of Non-Linear Integer Arithmetic for Smart Contract Verification. In Ruzica Piskac and Andrei Voronkov, editors, *Proc. of LPAR*, volume 94 of *EPiC Series in Computing*, pages 257–269. EasyChair, 2023.

[HHK$^+$20] Márton Hajdu, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Induction with Generalization in Superposition Reasoning. In Christoph Benzmüller and Bruce Miller, editors, *Proc. of CICM*, volume 12236 of *LNCS*, pages 123–137, Cham, 2020. Springer.

[HHK$^+$21] Márton Hajdu, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Inductive Benchmarks for Automated Reasoning. In Fairouz Kamareddine and Claudio Sacerdoti Coen, editors, *Proc. of CICM*, volume 12833 of *LNCS*, pages 124–129, Cham, 2021. Springer.

[HHK$^+$22] Márton Hajdu, Petra Hozzová, Laura Kovács, Giles Reger, and Andrei Voronkov. *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, volume 13660 of *LNCS*, chapter Getting Saturated with Induction, pages 306–322. Springer, Cham, 2022.

[HHKV21] Márton Hajdu, Petra Hozzová, Laura Kovács, and Andrei Voronkov. Induction with Recursive Definitions in Superposition. In Ruzica Piskac and Michael W. Whalen, editors, *Proc. of FMCAD*, pages 246–255. TU Wien Academic Press, 2021.

[HKH$^+$20] Martin Homola, Ján Kľuka, Petra Hozzová, Vojtěch Svátek, and Miroslav Vacura. Towards Higher-Order OWL. *KI-Künstliche Intelligenz*, 34(3):417–421, 2020.

[HKNV23] Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov. Program Synthesis in Saturation. In Brigitte Pientka and Cesare Tinelli, editors, *Proc. of CADE*, volume 14132 of *LNCS*, pages 307–324, Cham, 2023. Springer.

[HKR21] Petra Hozzová, Laura Kovács, and Jakob Rath. Automated Generation of Exam Sheets for Automated Deduction. In Fairouz Kamareddine and Claudio Sacerdoti Coen, editors, *Proc. of CICM*, volume 12833 of *Lecture Notes in Computer Science*, pages 185–196, Cham, 2021. Springer.

[HKRV22] Márton Hajdu, Laura Kovács, Michael Rawson, and Andrei Voronkov. The Vampire Approach to Induction. In *Proc. of PAAR*, 2022.

[HKV21] Petra Hozzová, Laura Kovács, and Andrei Voronkov. Integer Induction in Saturation. In André Platzer and Geoff Sutcliffe, editors, *Proc. of CADE*, volume 12699 of *LNCS*, pages 361–377, Cham, 2021. Springer.

[Hoz24] Petra Hozzová. Integrating Answer Literals with AVATAR for Program Synthesis. In Laura Kovács and Michael Rawson, editors, *Proc. of the 7th and 8th Vampire Workshop*, volume 99 of *EPiC Series in Computing*, pages 13–20. EasyChair, 2024.

[HQSW20] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling Enumerative and Deductive Program Synthesis. In *Proc. of PLDI*, PLDI 2020, page 1159–1174, New York, NY, USA, 2020. Association for Computing Machinery.

[HV09] Krystof Hoder and Andrei Voronkov. Comparing Unification Algorithms in First-Order Theorem Proving. In *Proc. of KI*, pages 435–443, 2009.

[IPP$^+$21] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. Cyclic Program Synthesis. In *Proc. of PLDI*, page 944–959, 2021.

[JS17] Susmit Jha and Sanjit A. Seshia. A Theory of Formal Synthesis via Inductive Learning. *Acta Informatica*, 54(7):693–726, 2017.

[KHDR21] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. Semantics-Guided Synthesis. *Proc. ACM Program. Lang.*, 5(POPL), 2021.

[KKKS13] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis Modulo Recursive Functions. In *Proc. of OOPSLA*, page 407–426, 2013.

[KKR+23] Konstantin Korovin, Laura Kovács, Giles Reger, Johannes Schoisswohl, and Andrei Voronkov. ALASCA: Reasoning in Quantified Linear Arithmetic. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Proc. of TACAS*, pages 647–665, Cham, 2023. Springer.

[Kle45] S.C. Kleene. On the Interpretation of Intuitionistic Number Theory. *J. Symbolic Logic*, 10:109–124, 1945.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*, volume 3. Springer, 06 2000.

[KP13] A. Kersani and N. Peltier. Combining Superposition and Induction: A Practical Realization. In *Proc. of FroCoS*, pages 7–22, 2013.

[KRV17] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to Terms with Quantified Reasoning. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proc. of POPL*, volume 52 of *ACM SIGPLAN Notices*, pages 260–270. ACM, 2017.

[KT14] Daniel Kroening and Michael Tautschnig. CBMC – C Bounded Model Checker. In Erika Ábrahám and Klaus Havelund, editors, *Proc. of TACAS*, pages 389–391. Springer, 2014.

[Kun96] Kenneth Kunen. The Semantics of Answer Literals. *J. of Automated Reasoning*, 17(1):83–95, 1996.

[KV13] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In N. Sharygina and H. Veith, editors, *Proc. of CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

[Lei10] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Proc. of LPAR*, pages 348–370. Springer, 2010.

[LM12] K. Rustan M. Leino and Aleksandar Milicevic. Program Extrapolation with Jennisys. In *Proc. of OOPSLA*, OOPSLA '12, page 411–430, 2012.

[LWC74] Richard C. T. Lee, Richard J. Waldinger, and Chin-Liang Chang. An Improved Program-Synthesizing Algorithm and Its Correctness. *Commun. ACM*, (4):211–217, 1974.

[MNnB+22] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. Bottom-up Synthesis of Recursive Functional Programs using Angelic Execution. *Proc. ACM Program. Lang.*, 6(POPL), 2022.

[MW80] Zohar Manna and Richard Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasonings*, volume I, pages 371–443. Elsevier and MIT Press, 2001.

[PCI+20] Grant Passmore, Simon Cruanes, Denis Ignatovich, David Aitken, Matthew Bray, Elijah Kagan, Konstantin Kanishev, Ewen Maclean, and Nicola Mometto. The Imandra Automated Reasoning System. In N. Peltier and V. Sofronie-Stokkermans, editors, *Proc. of IJCAR*, volume 12167 of *LNCS*, pages 464–471. Springer, 2020.

[PKSL16] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[PPR+21] Saswat Padhi, Elizabeth Polgreen, Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. The SyGuS Language Standard Version 2.1. `https://sygus.org/language/`, 2021.

[PS19] Nadia Polikarpova and Ilya Sergey. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.*, 3(POPL), 2019.

[Reg18] Giles Reger. Revisiting Question Answering in Vampire. *EPiC Series in Computing*, 53:64–74, 2018.

[RK15] Andrew Reynolds and Viktor Kuncak. Induction for SMT Solvers. In D. D'Souza, A. Lal, and K. G. Larsen, editors, *Proc. of VMCAI*, volume 8931 of *LNCS*, pages 80–98. Springer, 2015.

[Rob65] John A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[RSHR22] Michael Rawson, Martin Suda, Petra Hozzová, and Giles Reger. Reuse of Introduced Symbols in Automatic Theorem Provers. In Boris Konev, Claudia Schon, and Alexander Steen, editors, *Proc. of PAAR*, volume 3201 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.

[RSV18] Giles Reger, Martin Suda, and Andrei Voronkov. Unification with Abstraction and Theory Instantiation in Saturation-Based Reasoning. In *Proc. of TACAS*, pages 3–22, 2018.

[RSV21] Giles Reger, Johannes Schoisswohl, and Andrei Voronkov. Making Theory Reasoning Simpler. In J. F. Groote and K.G. Larsen, editors, *Proc. of TACAS*, volume 12652 of *LNCS*, pages 164–180. Springer, 2021.

[RV01] Tatiana Rybina and Andrei Voronkov. A Decision Procedure for Term Algebras with Queues. *ACM Transactions on Computational Logic*, 2(2):155–181, 2001.

[RV19] Giles Reger and Andrei Voronkov. Induction in Saturation-Based Proof Search. In P. Fontaine, editor, *Proc. of CADE*, volume 11716 of *LNCS*, pages 477–494. Springer, 2019.

[RV20] Giles Reger and Andrei Voronkov. Induction in Saturation-Based Proof Search. EasyChair Smart Slide, 2020.

[SDE12] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An Automated Prover for Properties of Recursive Data Structures. In C. Flanagan and B. König, editors, *Proc. of TACAS*, volume 7214 of *LNCS*, pages 407–421. Springer, 2012.

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From Program Verification to Program Synthesis. In *Proc. of POPL*, page 313–326, 2010.

[SGSM20] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *Proc. of CCS*, pages 621–640. ACM, 2020.

[SJB08] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching Concurrent Data Structures. In *Proc. of PLDI*, pages 136–148, 2008.

[SL09] Armando Solar-Lezama. The Sketching Approach to Program Synthesis. In *Proc. of APLAS*, pages 4–13, 2009.

[SST14] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A Cross-Community Infrastructure for Logic Solving. In *Proc. of IJCAR*, pages 367–373, 2014.

[STB$^+$06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial Sketching for Finite Programs. In *Proc. of ASPLOS*, pages 404–415, 2006.

[Sut16] G. Sutcliffe. The CADE ATP System Competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[Sut22] G. Sutcliffe. The Logic Languages of the TPTP World. *Logic Journal of the IGPL*, 2022.

[SWL$^+$94] Mark Stickel, Richard Waldinger, Michael Lowry, Thomas Pressburger, and Ian Underwood. Deductive Composition of Astronomical Software from Subroutine Libraries. In *Proc. of CADE*, pages 341–355, 1994.

[Tam95] Tanel Tammet. Completeness of Resolution for Definite Answers. *J. of Logic and Computation*, 5(4):449–471, 08 1995.

[TB13] Emina Torlak and Rastislav Bodik. Growing Solver-Aided Languages with Rosette. In *Proc. of Onward!*, Onward! 2013, page 135–152, 2013.

[TGD15] Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. Program Synthesis Using Dual Interpretation. In *Proc. of CADE*, pages 482–497, 2015.

[TNS$^+$21] Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. Example-Guided Synthesis of Relational Queries. In *Proc. of PLDI*, page 1110–1125, 2021.

[Vor14] Andrei Voronkov. AVATAR: The Architecture for First-Order Theorem Provers. In A. Biere and R. Bloem, editors, *Proc. of CAV*, volume 8559 of *LNCS*, pages 696–710. Springer, 2014.