DIPLOMARBEIT

Program Synthesis of Provable Recursive Functions

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Technische Mathematik

eingereicht von

Eva Maria Wagner, BSc
Matrikelnummer 01427565

ausgeführt am Institut für Logic and Computation
der Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuerin: Univ.Prof.in Dr.in techn. Laura Kovács, Msc
Mitwirkung: Projektass.in Mgr. Petra Hozzová

Wien, 14.05.2024 [redacted] [redacted]

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

TECHNISCHE
UNIVERSITÄT
WIEN

DIPLOMARBEIT

Program Synthesis of Provable Recursive Functions

ausgeführt am

Institute of
Logic and Computation
TU Wien

unter der Anleitung von

Univ.Prof.in Dr.in techn. Laura Kovács, MSc

durch

Eva Maria Wagner, BSc
Matrikelnummer: 01427565

Wien, am 13. Mai 2024

Kurzfassung

Wir studieren die Synthese von rekursiven Funktionen mithilfe eines sättigungsbasierten, automatisierten Theorembeweis-Tools. Wir verwenden das Superpositionsprinzip, um die Korrektheit der Spezifizierungen an die Funktionen zu beweisen und konstruieren währenddessen den Code, der genau diese Spezifizierung erfüllt. Die Spezifizierungen der Funktionen sind in Prädikatenlogik ausgedrückt mit induktiv definierten Datentypen. Wir stellen neue Folgerungsregeln für Induktion vor und verwenden sogenannte Beantwortungsliterale, um rekursive Funktionen aus der Beweisableitung zu synthetisieren. Wir zeigen die Herausforderungen von Synthetisierung rekursiver Funktionen anhand konkreter Beispiele und heben dabei unsere Lösungen hervor.

Abstract

We study the synthesis of recursive functions using saturation-based first-order theorem proving. We use superposition reasoning to prove the correctness of function specifications while constructing code that satisfies the given specification. The function specifications are expressed as first-order formulas with inductively defined data types. We present new inference rules for induction in saturation and use answer literals to synthesize recursive functions from saturation-based proof search. We show the challenges of recursive synthesis using concrete examples, highlighting our solutions in this context.

Acknowledgement

First and foremost I would like to express my gratitude to my supervisor Laura Kovács for her constant availability, facilitating weekly meetings that provided a platform for insightful discussions and constructive feedback as well as prompt corrections. She not only supervised my thesis but also invited me to numerous academic events to network and collaborate with professionals in the field. I am also very thankful to her for giving me the opportunity to attend the IJCAR 2024 conference in Nancy, France.

Secondly, I would like to thank Petra Hozzová who I have worked with over the past few months. Her availability for any questions is not to be taken for granted.

I would also like to thank my partner Fabian, who has supported me through the ups and downs of my studies. He helped me to overcome insecurities and self-doubts.

Last but not least I want to thank my older sister Susanne and my cousin Nora for their proofreading and constructive feedback which greatly contributed to the outcome of this thesis.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am 13. Mai 2024

Name der Autorin

Contents

1 Introduction **1**
- 1.1 Motivating Example . . . 3
- 1.2 Contributions and Outline . . . 6

2 Preliminaries **8**
- 2.1 Mathematical Background of Automating Proofs in Propositional Logic . . . 8
- 2.2 Mathematical Background of Automating Proofs in First-Order Logic . . . 11
  - 2.2.1 Preprocessing in First-Order Logic . . . 12
  - 2.2.2 Automating Proving in First-Order Logic . . . 15
- 2.3 First-Order Logic with Equality: The Superposition Calculus . . . 17

3 First Revisit of Motivating Example **22**

4 Synthesis of Non-recursive Programs in Saturation **26**
- 4.1 Tracking Changes in Proof Derivations with Answer Literals . . . 26
- 4.2 Illustrative Example: Maximum of Two Naturals . . . 30
- 4.3 Limitations . . . 32

5 Second Revisit of Motivating Example **34**
- 5.1 From Non-recursive to Recursive Synthesis: Changing Order of Quantifiers 34
- 5.2 Recursive Synthesis in Theory . . . 37
- 5.3 Recursive Synthesis in Practice using VAMPIRE . . . 38

6 Synthesis of Recursive Programs in Saturation **45**
- 6.1 Inductive Structures . . . 45
- 6.2 Constructing Recursive Functions from Inductive Proofs . . . 49
- 6.3 Magic Axioms for Different Inductive Structures . . . 51
  - 6.3.1 Natural Lists . . . 51
  - 6.3.2 Natural Binary Trees . . . 52

7 Exploring Recursive Synthesis in Saturation over Different Inductive Structures **53**
- 7.1 Recursive Synthesis over Natural Numbers . . . 53
- 7.2 Recursive Synthesis over Natural Lists . . . 61
- 7.3 Recursive Synthesis over Natural Binary Trees . . . 70

8 Related Work **78**

9 Conclusions **80**

Bibliography **82**

1 Introduction

Program synthesis is a research field in computer science that emerged in the 1950s with the establishment of artificial intelligence. The idea is that a computer can automatically derive a program that has been specified by a user in advance. The format of these requirements of the to-be synthesized code, that are also called specifications, depends on the framework that is used. Automating the process of generating code can significantly reduce time and errors, and, in the best case, leads to implementations of complex requirements or specifications that may be difficult for programmers to implement manually. Today, program synthesis is successfully used in many different research areas, such as software engineering, biological discovery, computer-aided education, or end-user programming. There are several applications of program synthesis in mass-market industrial products, e.g. FlashFill or GitHub Copilot, [GPS17].

With formal program verification correctness of code can be proven and is widely used in various industries. Here, the task is to firstly translate the code into formal semantics that the computer can take as input and secondly prove its validity with the help of a theorem prover, see Figure 1.1. Successful execution leads to code that has been proven to be correct, which means that no further testing is required.

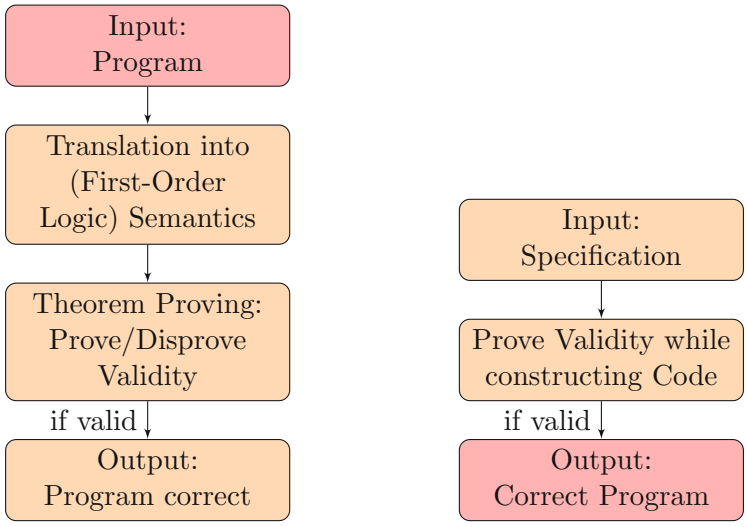
Deductive program synthesis or verified program synthesis combines these two ideas of formal program verification and program synthesis, see Figure 1.1. This leads to, not only *automatically constructing code* according to the given specification, but also simultaneously *proving* that the constructed code fulfills the given specification. Integrating theorem proving, which is a part of formal program verification, with program synthesis can achieve this.

The first step, in order to successfully synthesize code that is proven to be correct is to express the specification in a language that the to-be-used theorem prover understands. In our setting this is a *first-order logic* formula that has the following format

$$\forall\, \overline{x}\, \exists\, y. F[\overline{x}, y]. \tag{1.1}$$

In (1.1) the formula F specifies the *task* of the to-be synthesized program, the variables $\overline{x} = x_1, ..., x_n$ correspond to the *program input* and the variable y corresponds to the *program output*.

The task is to come up with an implementation that *fully automatically* (i) tracks all necessary information from the *fully automatic* proof and (ii) constructs the correct function from the extracted information for a given specification (1.1). In [HKNV23] Petra Hozzová, Laura Kovács, and Andrei Voronkov successfully implemented such a synthesis



Formal Program Verification **Verified Program Synthesis**

Figure 1.1: Comparing formal program verification with verified program synthesis.

device for the specific theorem prover Vampire.

Induction is an important proving technique that successfully enables theorem provers to prove more complex theorems, [HKRV22]. Induction is therefore also key for *synthesizing* more complex programs.

There is a one-to-one correspondence of induction and *recursive functions*, functions that call upon themselves. When induction is used during the proof, we exploit this correspondence to directly create a *recursive program*, which is a program that contains recursive functions. The successful implementation of synthesis that makes use of induction during an automatic proof is what we achieved in [HAH+24] which therefore enables synthesis of recursive programs.

In order to explain this process of synthesizing recursive functions from formal specifications in more detail, we will present an example in Section 1.1 that illustrates the mentioned correspondence of induction and recursion in a simple way. At a later stage, we will use the example in Section 1.1 to showcase how we can use this correspondence to fully automatize program synthesis of recursive functions. This includes

1. *manually proving* a specification using a formal framework in Chapter 3,
2. *manually synthesizing* a program from a given specification using a formal framework in Section 5.2,
3. and finally, *automatically synthesizing* a program from a given specification using an operating implementation in Section 5.3.

Steps 1 and 2 are my contributions in [HAH+24].

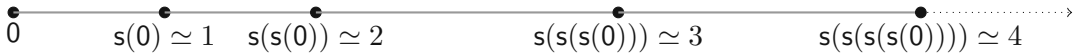
1.1 Motivating Example

We present an example that shows how to synthesize a recursive function from a given formal specification (1.1) by proving the specification by induction and tracking changes in the proof. Note here that we solely look at the informal proof for a better understanding of the correspondence of induction and recursion.

Before being able to express the formal specification, we need to establish the setting we are working with. In this case, it is the theory of natural numbers. Therefore we are equipped with the constructors $\mathsf{0}$ and s (the successor function s is more commonly known as the "+1" operator). The first constructor, $\mathsf{0}$, is a constant (i.e. a function of arity 0) and the second constructor, s, is a function of arity 1. The following axioms define these constructors.

$$\forall x.\ \mathsf{s}(x) \neq \mathsf{0} \tag{Nat1}$$
$$\forall x, y.\ \mathsf{s}(x) = \mathsf{s}(y) \to x = y \tag{Nat2}$$

This setting translates exactly to what we know of the set of natural numbers $\mathbb{N}$. The axioms (Nat1) and (Nat2) make sure of two things. First, there is no element coming before $\mathsf{0}$, and second, there are countably infinite elements coming after zero. This leads to an algebraic datatype $\mathbb{N}$ with the following structure:



What we will also need are the two commonly known operators " + " and "·". They are defined inductively over the theory of natural numbers by the following axioms.

$$\forall x \in \mathbb{N}.x + \mathsf{0} = x \tag{Add1}$$
$$\forall x, n \in \mathbb{N}.x + \mathsf{s}(n) = \mathsf{s}(x + n) \tag{Add2}$$
$$\forall x \in \mathbb{N}.x \cdot \mathsf{0} = \mathsf{0} \tag{Mult1}$$
$$\forall x, n \in \mathbb{N}.x \cdot \mathsf{s}(n) = x \cdot n + x \tag{Mult2}$$

These axioms give us the tools we need for expressing the *law of distributivity of natural numbers*. This law states that when two numbers are multiplied with the same factor and added, one can simplify it by first adding those two numbers and then multiplying them, e.g. $3 \cdot 4 + 3 \cdot 6 = 3 \cdot (4 + 6) = 3 \cdot 10$. Expressed in first-order logic, we have

$$\forall x_1, x_2, x_3 \in \mathbb{N}.\ x_1 \cdot x_2 + x_1 \cdot x_3 = x_1 \cdot (x_2 + x_3). \tag{Law of Distributivity}$$

We need to adapt (Law of Distributivity) in order to get a specification of the form (1.1). The goal is to synthesize a function that computes $x_2 + x_3$, this is what the output y should be. This then leads to the following input.

Input: $\forall x_1, x_2, x_3 \in \mathbb{N}.\ \exists y \in \mathbb{N}.\ x_1 \cdot x_2 + x_1 \cdot x_3 = x_1 \cdot y$ (SPD)

The goal of the synthesis task is then the following: by proving (SPD) we want to come up with a function $f(x_1, x_2, x_3)$ that computes y, which is $x_2 + x_3$. When the task is to only prove (SPD) one needs to, in theory, only prove the existence of element y but does not need to come up with how y looks like. However, we can also use the proof of existence of y to construct y. This describes the core of the combination of theorem proving and synthesis: we want to not only have the information of validity of (SPD) but also construct a program that returns the specific output y for given input x.

Now we prove (SPD) on paper and use the important proving technique of induction. Here, we are using induction over the natural numbers. Informally described, induction is like constructing an infinite ladder. Firstly, we prove that a statement holds for the base element $\mathsf{0}$, and then we prove that, if the statement holds for an arbitrary element n, it also holds for the element that is coming after, $\mathsf{s}(n)$. If this can be done we can deduce by the nature of the natural numbers that the statement holds for all elements in $\mathbb{N}$. When expressing induction in first-order logic, we have

$$(F[\mathsf{0}] \wedge \forall n.(F[n] \rightarrow F[\mathsf{s}(n)])) \rightarrow \forall x.F[x]. \tag{IndAx}$$

We call the formula (IndAx) the induction axiom. It holds for arbitrary first-order logic formulas F.

The idea of our synthesis task is that when we apply induction while proving the input specification (SPD), we also construct a recursive function that stores the following information:

1. the input argument on which induction is applied,
2. the output for the input $\mathsf{0}$, the base case,
3. and the output for the input $\mathsf{s}(n)$, which makes use of the output for the input n.

We will explain this in more detail by writing down a proof of (SPD).

Proof. We apply induction on the formula

$$F_{dis}[x] : \forall x_1, x_2\ \exists y.\ x_1 \cdot x_2 + x_1 \cdot x = x_1 \cdot y.$$

By doing this, we firstly show that

$$F_{dis}[\mathsf{0}] : \forall x_1, x_2\ \exists y.\ x_1 \cdot x_2 + x_1 \cdot \mathsf{0} = x_1 \cdot y$$

holds. Because of (Add1) and (Mult1) the term $x_1 \cdot x_2 + x_1 \cdot \mathsf{0}$ simplifies to $x_1 \cdot x_2$. The formula

$$\forall x_1, x_2.\ x_1 \cdot x_2 = x_1 \cdot x_2$$

is trivially true, therefore setting $y_0 := x_2$ leads to validity of $F_{dis}[0]$.

As a next step we assume that

$$F_{dis}[n] : \forall x_1, x_2 \in \mathbb{N}. \exists y \in \mathbb{N}. x_1 \cdot x_2 + x_1 \cdot n = x_1 \cdot y \tag{IndHyp}$$

holds and show that under the assumption of $F_{dis}[n]$ also $F_{dis}[\mathsf{s}(n)]$ holds. The calculation (1.2) shows that one can set $y_{\mathsf{s}} := \mathsf{s}(y)$.

$$x_1 \cdot x_2 + x_1 \cdot \mathsf{s}(n) \overset{\text{(Mult2)}}{=} x_1 \cdot x_2 + x_1 \cdot n + x_1 \overset{\text{(IndHyp)}}{=} x_1 \cdot y + x_1 \overset{\text{(Mult2)}}{=} x_1 \cdot \mathsf{s}(y) \tag{1.2}$$

We see that we have found an element y_{s} such that $F_{dis}[\mathsf{s}(n)]$ holds. Therefore $F_{dis}[\mathsf{s}(n)]$ is valid under assumption $F_{dis}[n]$. We have shown that the induction premises in (IndAx) hold and can conclude that

$$\forall x. F_{dis}[x],$$

which is exactly the specification (SPD) we wanted to prove. □

Going back to the three bullet points above, we can construct a recursive function f that has

1. variable x_3 as input, corresponding to the application of induction on exactly this variable,
2. the output x_2 for the input $\mathsf{0}$ and
3. the output $\mathsf{s}(y)$ for the input $\mathsf{s}(n)$, where $y = f(n)$.

In summary, the constructed program is

Output: $f(x_3)$, (1.3)

where f is defined by

$$\begin{aligned} f(\mathsf{0}) &= x_2 \\ f(\mathsf{s}(n)) &= \mathsf{s}(f(n)). \end{aligned} \tag{1.4}$$

We can easily verify that program (1.3) where the recursive function f is defined by (1.4) has exactly the intended behavior of the given specification (SPD). It adds x_3-times "+1" to x_2, in other words $x_2 + x_3$.

To sum up, we have proven that (1.3) is the correct program corresponding to the specification (SPD).

1.2 Contributions and Outline

Outline. In this thesis, I show how the process discussed in Section 1.1 is formalized, leading to *fully automated synthesis of provable recursive programs.*

For this, we firstly explain the basic notions of the resolution calculus in propositional and first-order logic and further extend this to superposition first-order theorem proving in Chapter 2. The superposition calculus is used by Vampire which enables reasoning over theories with equality, as we have seen in 1.1 where we have reasoned over the natural numbers. We give a derivation for the motivating example of Section 1.1 in Chapter 3 for a better understanding of the superposition calculus.

As a second step, in Chapter 4 we introduce the concept of answer literals and show how they can synthesize non-recursive functions using the superposition calculus. This is the setting that essentially automatizes the process in Section 1.1, extracting the information about variable y in the proof. We further discuss limitations of this framework related to recursive synthesis.

At this stage, we revisit our motivating example a second time to show how the non-recursive synthesis framework can be extended to synthesize the program (1.3). We also show how this works in practice using the first-order theorem prover Vampire, see Chapter 5.

In Chapter 6 we introduce the notion of inductive structures and prove certain properties to obtain a uniqueness relation of recursive functions and induction. We formalize the extension of recursive synthesis explained in Chapter 5 by using specially defined induction axioms dubbed as *magic axioms.*

We finally give several examples in Chapter 7 for different inductive structures (natural numbers, natural lists, and natural binary trees) that show how this simple framework can be extended further in order to synthesize more complex programs.

Finally, we compare different synthesis frameworks with ours, in Chapter 8.

My contributions. During the development of a recursive synthesis method [HAH+24], implemented in the theorem prover Vampire, my contributions were the following:

1. creating interesting and challenging specifications to synthesize,
2. figuring out how the developed framework can synthesize these specifications by using induction and
3. improving the developed framework in such a way that it can synthesize programs it was not able to synthesize beforehand.

My contributions deepened the understanding of recursive synthesis using the superposition calculus and therefore helped to improve the implementation of the developed synthesis approach.

Paper acceptance at the IJCAR 2024 conference. Parts of this thesis contributed to the paper "*Synthesis of Recursive Programs in Saturation*" [HAH+24] that was accepted at IJCAR 2024 (International Joint Conference on Automated Reasoning). The paper was mainly written by Petra Hozzvá and supervised by Laura Kovács and Andrei Voronkov. Part of the implementation was done by Daneshvar Amrollahi. I lead the efforts in designing challenging examples to be used for synthesis and proving each example in the superposition calculus.

2 Preliminaries

In this chapter, the basic concept of first-order theorem proving is discussed. In Section 2.1 and Section 2.2 we recall and follow results from [GGH18], whereas Section 2.3 builds upon the content of chapter 7, [NR01], and chapter 10, [DV01], of the *Handbook of Automated Reasoning*.

The goal of first-order theorem proving is to establish validity of, hence prove, an arbitrary statement that is formulated in first-order logic. The underlining proving principle used here, as well as in this thesis is *proof by refutation*. There, it is assumed that the opposite of a given statement ϕ is true with the goal of finding a contradiction. When this can be done, the original statement ϕ must hold. In terms of logic, this can be written as

$$\phi \text{ is valid } \iff \neg\phi \text{ is unsatisfiable,}$$

which means that, when a contradiction of statement $\neg\phi$ can be found, it immediately follows that ϕ must hold. But how does one prove unsatisfiability of a given formula? The keyword here is *resolution*. The input formula is resolved until the empty clause, denoted as $\square$, is derived. The empty clause $\square$ is exactly the contradiction that is looked for, it proves unsatisfiability of the negated input statement $\neg\phi$, and therefore concludes the proof of validity of ϕ. To understand the principle of resolution in more detail, firstly the simpler resolution framework in propositional logic is presented in Section 2.1 and afterward extended to first-order logic in Section 2.2. In Section 2.3 we introduce the superposition calculus for reasoning over theories with equality.

It is assumed that the reader fairly knows the basic notions of propositional and first-order logic, more detailed syntactical and semantic definitions can be found in Chapter II and Chapter III of [GGH18].

2.1 Mathematical Background of Automating Proofs in Propositional Logic

Propositional logic consists of boolean variables $p, q, r, s, ...$, the connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, as well as $\top, \bot$ (verum and falsum). The set of boolean variables is denoted by $\mathcal{BV}$. In the following the process of proving validity of a propositional formula ϕ by resolution is described. It consists of two steps, (i) preprocessing and (ii) proving the formula ϕ.

Definition 2.1.1 (Validity of Propositional Formulas). A formula ϕ in propositional logic is said to be *valid* if and only if

$$\forall b : \mathcal{BV} \rightarrow \{\top, \bot\}.\ \bar{b}(\phi) = \top.$$

A valid propositional formula is also called a *tautology*.

Note here that b is a function that assigns for each boolean variable occurring in ϕ either the value true ($\top$) or false ($\bot$). We call these types of functions *assignments*. The function $\overline{b}$ is an extension of b, in the sense that it can evaluate the truth values of propositional formulas. This is done by using the truth values for the variables specified by b and nextly interpreting the meaning of the connectives $\neg, \wedge, \vee, \rightarrow$, and $\leftrightarrow$.

We show a simple example of evaluating a propositional formula for an assignment b in the following.

Example 2.1.2 Let the assignment b be defined by $b(p) = \top$, $b(q) = \top$ and $b(r) = \bot$ for the boolean variables p, q and r. Then the extension $\overline{b}$ evaluates the input formula $\phi_1 : (\neg p \vee q) \wedge r$ as

$$\overline{b}(\phi_1) = ((\neg\top) \vee \top) \wedge \bot = (\bot \vee \top) \wedge \bot = \top \wedge \bot = \bot .$$

We conclude that the formula ϕ_1 evaluates to false under assignment b.

Let us look at an example of a valid propositional formula, so formulas that evaluate to true for *every* assignment b.

Example 2.1.3 The formula $\phi_2 : (p \wedge \neg p) \rightarrow q$ is valid. The two possible assignments for variable p are $b_1(p) = \top$ and $b_2(p) = \bot$ (the assigned value for variable q does not matter in this case), from which $\overline{b}_1(\phi_2) = \overline{b}_2(\phi_2) = \top$ follows.

Let us now go back to the initial task, establishing validity of propositional formulas. We start with preprocessing the input formula ϕ.

Preprocessing. The first step for proving validity of a given propositional formula ϕ is to transform its negation $\neg\phi$ into a *logically equivalent* formula, that is in *conjunctive normal form*.

Definition 2.1.4 (Logical Equivalence in Propositional Logic). Two formulas ϕ and ϕ' in propositional logic are called *(logically) equivalent* if and only if

$$\forall b : \mathcal{BV} \rightarrow \{\top, \bot\}.\ \overline{b}(\phi) = \overline{b}(\phi').$$

Definition 2.1.5 (Conjunctive Normal Form). A propositional formula ϕ_{cnf}^{pro} is said to be in *conjunctive normal form, CNF* for short when it has the following format

$$\phi_{cnf}^{pro} := \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} L_{i,j}, \tag{2.1}$$

where $L_{i,j}$ are literals, so either negated (e.g. $\neg p$) or non-negated (e.g. p) boolean variables. For each i, the expression $\bigvee_{j=1}^{m_i} L_{i,j}$ is called a *clause*, that is, a disjunction of literals. A clause that contains no literals is called the *empty clause* and is denoted as $\square$. When there is an index i such that $m_i = 0$, the empty clause is contained in ϕ_{cnf}^{pro}, which makes ϕ_{cnf}^{pro}

automatically unsatisfiable. A formula in CNF is also called a set of clauses and can be written as $\phi_{cnf}^{pro} = \{C_1, ..., C_n\}$, where $C_i = \{L_{i,1}, ..., L_{i,m_i}\}$ for $i \in \{1, ..., n\}$.

Transformation of $\neg\phi$ into CNF is called *preprocessing* and is done in several steps that will be explained here.

As a first step, the connectives $\rightarrow$ and $\leftrightarrow$ are removed. This can be done with the following equivalence transformations

$$\begin{aligned} \phi \rightarrow \psi &\iff \neg\phi \vee \psi \\ \phi \leftrightarrow \psi &\iff (\neg\phi \vee \psi) \wedge (\neg\psi \vee \phi). \end{aligned}$$

When the formula ϕ only contains connectives $\neg$, $\wedge$ and $\vee$ the next step is to move all outer negations directly next to the variables using De Morgan's Laws and the rule of double negation

$$\begin{aligned} \neg(\phi \vee \psi) &\iff \neg\phi \wedge \neg\psi \\ \neg(\phi \wedge \psi) &\iff \neg\phi \vee \neg\psi \\ \neg(\neg\phi) &\iff \phi. \end{aligned}$$

Then using the distributivity laws

$$\begin{aligned} (\phi \wedge \psi) \vee \chi &\iff (\phi \vee \chi) \wedge (\psi \vee \chi) \\ \chi \vee (\phi \wedge \psi) &\iff (\chi \vee \phi) \wedge (\chi \vee \psi), \end{aligned}$$

the formula can be transformed into CNF.

Resolution calculus. The second step of proving validity of a propositional formula ϕ, after transforming its negation into CNF, is to derive from the given set of clauses (that is exactly $\neg\phi$) the empty clause $\square$ using the resolution and factoring rule, see Figure 2.1. When the empty clause using the resolution and the factoring rule can be derived from $\neg\phi$, it is said that there is a resolution refutation of $\neg\phi$. The following theorem assures that the introduced resolution calculus is sound and complete.

Theorem 2.1.6 (Soundness and completeness of propositional resolution). *Let ϕ be a propositional formula. Then the following holds.*

$$\textit{Soundness: } \textit{If there is a resolution refutation of } \neg\phi, \textit{ then } \phi \textit{ is valid.} \tag{2.2}$$

$$\textit{Completeness: } \textit{If } \phi \textit{ is valid, then there is a resolution refutation of } \neg\phi. \tag{2.3}$$

Proof. (2.2) follows from II.3.4 of [GGH18], whereas (2.3) follows from II.3.11 of [GGH18]. $\square$

The proving process then works like this. Each application of an inference rule results in a new clause. Assuming that $\neg\phi$ holds, each newly received clause must also hold (soundness). Therefore the derived clause can be added to the input clause set and further clauses can be derived. When at some point the empty clause $\square$ is added to the clause set, the contradiction has been found (completeness). The empty clause $\square$ can never hold,

Binary resolution (BR):	**Factoring (F):**
$\dfrac{\underline{L} \vee C \quad \underline{\neg L} \vee D}{C \vee D}$	$\dfrac{\underline{L} \vee \underline{L} \vee C}{L \vee C}$

Figure 2.1: Inference rules of binary resolution and factoring in propositional logic. Expression L denotes a literal, and expressions C and D denote clauses. Underlining literals is a notation to clarify which literals are selected for resolution or factoring.

it is a disjunction of literals, to be true at least one literal must be fulfilled. Because it is empty there is no literal that can be fulfilled, hence it is false. On the other hand, our proof derivation states that the empty clause must hold. Because of the soundness of our calculus, from Figure 2.1, the formula $\neg\phi$ is unsatisfiable, hence ϕ must be valid. Let us look at a short example.

Example 2.1.7 We want to prove validity of the propositional formula $\phi_3 : (p \wedge \neg p) \rightarrow (q \vee r)$. For this, we firstly transform $\neg\phi_3$ into CNF, that is

$$\neg((p \wedge \neg p) \rightarrow (q \vee r))) = \neg(\neg(p \wedge \neg p) \vee (q \vee r)) = p \wedge \neg p \wedge (q \vee r).$$

So $\neg\phi_3$ consists of the three clauses $C_1 : p$, $C_2 : \neg p$ and $C_3 : q \vee r$. Now one can apply binary resolution on clauses C_1 and C_2,

$$\frac{p \quad \neg p}{\square}\ \text{BR},$$

which results in the empty clause $\square$. This completes the proof; the formula ϕ_3 is indeed valid.

2.2 Mathematical Background of Automating Proofs in First-Order Logic

Having established a sound and complete calculus in propositional logic, Figure 2.1 is nextly extended to first-order logic. Here the setting consists of variables $x, y, z, ...$, constants $a, b, c, ...$, predicate symbols with arity $p, q, r, ...$, and functional symbols with arity $f, g, h, ...$ Expressions consisting of variables, constants, and functional symbols are called *terms*. For example using functional symbols f with arity 2, g with arity 3, and constant symbol a, we get the term $g(f(x), y, a)$. For given terms $t_1, ..., t_n$ and a predicate symbol p of arity n, expressions of the form $p(t_1, ..., t_n)$ are called *atomic formulas*. A formula consists of atomic formulas connected with $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$. Additionally, the *universal quantifier* $\forall$ and the *existential quantifier* $\exists$ can be used to bound variables.

The concept of validity in first-order logic changes slightly.

Definition 2.2.1 (Validity of First-Order Formulas). We say that a formula ϕ in first-order logic is *valid*, if and only if

$$\forall \mathcal{I}.\mathcal{I} \models \phi.$$

Here, $\mathcal{I}$ denotes an interpretation, that is, informally described, a mapping that gives the intended meaning of predicate symbols and functional symbols in our underlying language.

In this section, the process to prove validity of arbitrary closed formulas in first-order logic is presented. Note here that a formula ϕ is *closed* if every variable occurring in ϕ is bounded by either a universal ($\forall$) or an existential quantifier ($\exists$).

The two main steps described in Section 2.1, preprocessing of the input formula into CNF, and finding a resolution refutation are adjusted and formulated in the following.

2.2.1 Preprocessing in First-Order Logic

Preprocessing consists of transforming the negation of an arbitrary first-order logic formula that is closed into CNF.

Definition 2.2.2 (Conjunctive Normal Form). A first-order logic formula is said to be in *conjunctive normal form, CNF* for short when it has the following format

$$\phi_{cnf}^{fol} := \forall x_1 ... \forall x_n \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} L_{i,j}, \tag{2.4}$$

where $L_{i,j}$ are literals, so either negated or non-negated atomic formulas, and $x_1, ..., x_n$ are all variables occurring in ϕ_{cnf}^{fol}.

Transformation into CNF is done in several steps, which are explained next.

1. Cleansing of the formula. When looking at a formula of the form (2.4), all its quantifiers are located at the front. In order to move quantifiers to the front without changing the intended meaning of the input formula, it may be necessary to rename some variables, a process that is called *cleansing*.

Definition 2.2.3 (Cleansing). A first-order logic formula ϕ is called *cleansed* if

1. each variable occurring in ϕ is only bounded by one quantifier
2. no variable in ϕ is bounded by a quantifier in one part and is occurring as a free variable in another.

The following lemma holds.

Lemma 2.2.4 *Every first-order logic formula ϕ can be transformed into a logically equivalent cleansed formula ϕ.*

Proof. See IV.5.9 in [GGH18] □

Let us look at an example.

Example 2.2.5 The formula $\phi : (\forall x.\exists y.p(f(x), y)) \wedge (\exists x.g(x, c))$ can be transformed into the logically equivalent and cleansed formula $\phi_{clean} : (\forall x.\exists y.p(f(x), y)) \wedge (\exists z.g(z, c))$.

2. Transformation into prenex-normal form. After the formula has been cleansed, all its quantifiers can be moved to the front by using the following equivalence transformations:

$$\begin{aligned}
(Qx.\phi) \wedge \psi &\iff Qx.(\phi \wedge \psi) \\
\phi \wedge (Qx.\psi) &\iff Qx.(\phi \wedge \psi) \\
(Qx.\phi) \vee \psi &\iff Qx.(\phi \vee \psi) \\
\phi \wedge (Qx.\psi) &\iff Qx.(\phi \vee \psi) \\
(Qx.\phi) \rightarrow \psi &\iff \overline{Q}x.(\phi \rightarrow \psi) \\
(\phi \rightarrow (Qx.\psi)) &\iff Qx.(\phi \rightarrow \psi) \\
\neg(Qx.\phi) &\iff \overline{Q}x.\neg\phi.
\end{aligned}$$

Note here that the symbol Q denotes either a existential or universal quantifier, and $\overline{Q} = \forall$ if $Q = \exists$, whereas $\overline{Q} = \exists$ if $Q = \forall$.

When a first-order formula has all of its quantifiers in front, it is said to be in *prenex normal form, PNF* for short. Note here that this transformation is not necessarily unique, the sequence of the quantifiers can differ. But it is usually done in such a way that the existential quantifiers are as far ahead as possible, which will become clearer in the next step.

The following property of transformation into PNF can be stated.

Lemma 2.2.6 *Every first-order logic formula ϕ can be transformed into a logically equivalent formula ϕ_{pnf} that is in PNF.*
Proof. See in IV.5.11 [GGH18]. □

3. Skolemization. After the input formula is transformed into PNF, the next step is to remove all existential quantifiers by introducing new functional symbols or constants, so-called *skolem functions* or *skolem constants*. An important thing to note here is that in all the other preprocessing steps, the formula is transformed into a logically equivalent formula. A formal definition of this is the following.

Definition 2.2.7 (Logical Equivalence). Two formulas ϕ, ϕ' in first-order logic are said to be *logically equivalent*, $\phi \leftrightarrow \phi'$, if and only if

$$\forall \mathcal{I}.(\mathcal{I} \models \phi \iff \mathcal{I} \models \phi')$$

This means, essentially, that each interpretation that fulfills one formula must also fulfill the other. Looking at Skolemization, this is no longer the case. Here the transformation only ensures so-called *equisatisfiability*, which is defined like this.

Definition 2.2.8 (Equisatisfiability). Two formulas ϕ, ϕ' in first-order logic are said to be *equisatisfiable*, $\phi \leftrightarrow_{sat} \phi'$, if and only if

$$\exists \mathcal{I}.\mathcal{I} \models \phi \iff \exists \mathcal{I}'.\mathcal{I}' \models \phi'$$

The difference to logical equivalence is, that the interpretations no longer must be the same and, more importantly, that the property of validity is not necessarily preserved. But, since our theorem proving goal is to prove unsatisfiability, equisatisfiability suffices. If the transformed formula $\mathsf{cnf}(\neg\phi)$ can be proven to be unsatisfiable, it can *still* be concluded that the original formula $\neg\phi$ is also unsatisfiable, which in turn proves validity of ϕ. Let us look at Skolemization in more detail.

Lemma 2.2.9 *Let $\phi : \forall x_1, ..., \forall x_n \exists y.\phi'$ be a closed first-order formula and f a new functional symbol of arity n that is not occurring in ϕ. Then*

$$\forall x_1, ..., \forall x_n \exists y.\phi' \text{ is equisatisfiable to } \forall x_1, ..., \forall x_n.\phi'[y/f(x_1, ..., x_n)],$$

where f is a skolem function. *Note here that when $\phi : \exists y.\phi'$, the functional symbol f is of arity 0, in other words, a constant and therefore non-dependent on any other variables. If that is the case, f is called a* skolem constant.

Proof. See IV.5.13 in [GGH18]. □

Theorem 2.2.10 (Skolemization). *Let ϕ be a closed first-order logic formula in PNF. Then there exists a skolemized formula $\phi_{sko} : \forall x_1, ..., \forall x_n \phi'$, where ϕ' is quantifier-free such that*

$$\phi \leftrightarrow_{sat} \phi_{sko}.$$

Proof. See IV.5.14 in [GGH18]. □

4. Transformation into CNF. After the above preprocessing steps, the output formula is of the form $\phi : \forall x_1, ..., x_n.\phi'$, where ϕ' is quantifier-free. Therefore, ϕ' can be treated like a propositional formula (the atomic formulas act like boolean variables) and hence be transformed into CNF using the same process as described in Section 2.1. When this is done, all occurring variables in ϕ_{cnf} must be universally quantified because we started off with a closed formula, and all existential quantifiers were removed in the process of skolemization. Therefore all the quantifiers in front can be removed without a loss of information. This is done for notational reasons, the variables are still universally bounded, it is just not written down explicitly in the proof derivation. Therefore the final input after preprocessing will be the quantifier-free fragment of ϕ, which is ϕ'.

2.2.2 Automating Proving in First-Order Logic

As a next step, after transforming the formula ϕ into CNF, the goal is to come up with a similar resolution calculus as described in Section 2.1. The important difference here is, that in contrast to propositional logic, we are not working with boolean variables, that can be either equal or non-equal, but atomic formulas. So the question here is not if two given atomic formulas are equal, but if they can be transformed into the same equal formula and then be resolved. This transformation of two atomic formulas is done with *substitutions*.

Definition 2.2.11 (Substitution). We call a function $\theta : \{x_1 \mapsto t_1, ..., x_n \mapsto t_n\}$ a *substitution*, where $x_1, ..., x_n$ are variables, $t_1, ..., t_n$ terms and $n \geq 0$. The set $\mathsf{dom}(\theta) := \{x_1, ..., x_n\}$ is called the *domain of substitution* θ. Note that the identity function, denoted with id, is also a substitution.

Substitutions can be applied on any expression in first-order logic, meaning terms, literals, and formulas. For an expression E and substitution $\theta : \{x_1 \mapsto t_1, ..., x_n \mapsto t_n\}$ it is also written $E\theta$ for $E[x_1/t_1, ..., x_n/t_n]$. Note here that, when handling a formula ϕ in CNF, all variables occurring in ϕ are bounded by a universal quantifier. This means that when applying a substitution θ on a clause C, the clause $C\theta$ still holds under assumption C by the definition of universal quantifiers.

Definition 2.2.12 (Union and Composition of Substitutions). For two substitutions $\sigma_1 : \{x_1 \mapsto t_1, ..., x_n \mapsto t_n\}$ and $\sigma_2 : \{y_1 \mapsto s_1, ..., y_m \mapsto s_m\}$, where $\mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) = \emptyset$, the substitution $\sigma_1 \cup \sigma_2 := \{x_1 \mapsto t_1, ..., x_n \mapsto t_n, y_1 \mapsto s_1, ..., y_m \mapsto s_m\}$ is called the *union* of σ_1 and σ_2. For two substitutions $\theta : \{x_1 \mapsto t_1, ..., x_n \mapsto t_n\}$ and $\sigma : \{y_1 \mapsto s_1, ..., y_m \mapsto s_m\}$, where $\sigma = \sigma_1 \cup \sigma_2$ and $\mathsf{dom}(\sigma_1) \subseteq \mathsf{dom}(\theta)$ and $\mathsf{dom}(\sigma_2) \cap \mathsf{dom}(\theta) = \emptyset$, the *composition* of θ and σ is defined as $\theta \circ \sigma := \{x_1/t_1\sigma, ..., x_n/t_n\sigma\} \cup \sigma_2$. When applying the composition $\theta \circ \sigma$ on an expression E one can write $E(\theta \circ \sigma) \simeq (E\theta)\sigma$.

Definition 2.2.13 (Unifier). For two expressions E_1, E_2 we call a substitution θ a *unifier* of E_1 and E_2 if $E_1\theta \simeq E_2\theta$.

Remark 2.2.14 In our setting, the expressions E_1 and E_2 will be terms or literals. The intended meaning of $E_1 \simeq E_2$ is, that expressions E_1 and E_2 consist of the same symbols occurring at the same positions.

A unifier of two expressions is not necessarily unique. Therefore the question arises which substitution is best to use. We define an ordering over the set of all unifiers.

Definition 2.2.15 (Ordering of Unifiers). For two expressions E_1, E_2 and two unifiers θ_1, θ_2 of E_1, E_2 an *ordering* can be defined, where $\theta_1 \leq \theta_2$ if and only if there exists a substitution σ such that $\theta_1 \circ \sigma = \theta_2$.

Note here that for $\theta_1 = \theta_2$, the substitution σ is just the identity, id.

The idea of the ordering is that one unifier is considered to be smaller than the other when it can already achieve to unify the two literals without "needing" the substitution σ. There also exists a minimal element on the set of all unifiers of two expressions E_1 and E_2, if this set is non-empty.

Binary resolution (BR):	**Factoring (F):**
$\dfrac{\underline{L} \vee C \quad \underline{\neg L'} \vee D}{(C \vee D)\theta}$	$\dfrac{\underline{L} \vee \underline{L'} \vee C}{(L \vee C)\theta}$
where $\theta := \mathsf{mgu}(L, L')$.	where $\theta := \mathsf{mgu}(L, L')$.

Figure 2.2: Extension of the resolution calculus in first-order logic. Expressions L, L' denote literals and expression C and D are clauses.

Definition 2.2.16 (mgU). For two expressions E_1, E_2, the unifier θ is called the *most general unifier of* E_1 *and* E_2, if for every unifier θ' of E_1 and E_2 it already follows that $\theta \leq \theta'$. This can also be denoted as $\theta = \mathsf{mgU}(E_1, E_2)$.

Theorem 2.2.17 (Existence of mgU). *If two expressions* E_1 *and* E_2 *have a unifier then there also exists the most general unifier of* E_1 *and* E_2.
Proof. See IV.5.23 in [GGH18]. □

Let us consider the following example.

Example 2.2.18 The substitutions $\theta_1 : \{x_1 \mapsto g(x_3), x_4 \mapsto c, x_5 \mapsto g(x_2)\}$ and $\theta_2 : \{x_1 \mapsto g(g(c)), x_2 \mapsto c, x_3 \mapsto g(c), x_4 \mapsto c, x_5 \mapsto g(c)\}$ are both unifiers for the atomic formulas $A_1 : p(f(x_1, c), g(x_2))$ and $A_2 : p(f(g(x_3), x_4), x_5)$, but $\theta_1 \circ \sigma = \theta_2$ for $\sigma : \{x_2 \mapsto c, x_3 \mapsto g(c)\}$. Therefore the inequality $\theta_1 \leq \theta_2$ holds. Additionally, it holds that $\theta_1 = \mathsf{mgU}(A_1, A_2)$.

Remark 2.2.19 The first unification algorithm was developed by Robbinson in 1965, [Rob65], which firstly determines for two given expressions E_1 and E_2 whether they are unifiable or not and secondly returns the most general unifier $\mathsf{mgU}(E_1, E_2)$ if they are unifiable.

Resolution in first-order logic. With this setting, the resolution calculus for closed first-order formulas is completed. The rules in Figure 2.1 can be adapted by resolving clauses with literals that can be unified, see Figure 2.2. The principle stays the same. Firstly the input formula ϕ is negated and transformed into CNF. Secondly, the inference rules in Figure 2.2 are applied on the clauses until the empty clause can be derived. This completes the proof of validity of ϕ. Now, similarly to Section 2.1 the following theorem can be stated.

Theorem 2.2.20 (Soundness and completeness of first-order resolution). *Let* ϕ *be a first-order logic formula. Then the following holds.*

$$\textit{Soundness:} \quad \textit{If there is a resolution refutation of } \neg\phi, \textit{ then } \phi \textit{ is valid.} \tag{2.5}$$

$$\textit{Completeness:} \quad \textit{If } \phi \textit{ is valid, then there is a resolution refutation of } \neg\phi. \tag{2.6}$$

Proof. Check IV.5.28 for (2.5) and IV.5.33 for (2.6) in [GGH18]. □

Let us look once more at a small example.

Example 2.2.21 We prove validity of $\phi : (\exists x.p(f(x), x)) \rightarrow (\exists x.q(x) \vee \neg q(a))$. We transform $\neg\phi$ into CNF.

$$\begin{aligned}
\neg\phi &\overset{Cleansing}{=} \neg((\exists x.p(f(x), x)) \rightarrow (\exists y.q(y) \vee \neg q(a))) \\
&\overset{PNF}{=} \exists x \forall y.\neg(p(f(x), x) \rightarrow (q(y) \vee \neg q(a))) \\
&\overset{Skolemization}{=} \forall y.\neg(p(f(\sigma_x), \sigma_x) \rightarrow (q(y) \vee \neg q(a))) \\
&\overset{CNF1}{=} \forall y.\neg(\neg p(f(\sigma_x), \sigma_x) \vee (q(y) \vee \neg q(a))) \\
&\overset{CNF2}{=} \forall y.p(f(\sigma_x), \sigma_x) \wedge \neg q(y) \wedge q(a)
\end{aligned}$$

The term σ_x denotes the skolem constant for the existentially bounded variable x. We obtain a clause set that contains the clauses $C_1 : p(f(\sigma_x), \sigma_x)$, $C_2 : \neg q(y)$ and $C_3 : q(a)$. We apply binary resolution on C_2 and C_3 using the most general unifier $\theta : \{y \mapsto a\}$,

$$\frac{q(a) \quad \neg q(y)}{\square}\ \text{BR},$$

which results in the empty clause □. This completes the validity proof of ϕ.

2.3 First-Order Logic with Equality: The Superposition Calculus

In Section 2.2 the function and predicate symbols are stated, without necessarily knowing what they are supposed to do or which property they are supposed to verify. This only becomes clear when defining an interpretation, which gives intended meaning to symbols. Another possibility to describe the properties of functions and predicates is to use axioms, which are closed formulas in first-order logic. We already encountered this in the motivating example 1 where we translated the meaning of addition and multiplication into first-order logic formulas. Implicitly it was also described what the equality sign was supposed to do. This needs to be formally clarified by additional axioms which is done in this section.

We call a set of axioms together with the functional and predicate symbols a *theory*, $\mathcal{T}$. From now on, when wanting to establish validity of a given formula in first-order logic, it is always done with respect to their underlying theory.

Definition 2.3.1 (Validity of First-Order Formula wrt. Underlying Theory). A formula ϕ is *valid wrt. a theory* $\mathcal{T}$ if and only if for each interpretation $\mathcal{I}$ the implication

$$\mathcal{I} \models \mathcal{A} \rightarrow \mathcal{I} \models \phi$$

holds, where $\mathcal{A} = \{A_1, .., A_n\}$ denotes the set of axioms from theory $\mathcal{T}$.

When a function or predicate symbol is described by a set of axioms it is said to be *interpreted*. One example of an interpreted predicate symbol is the equality sign. It is clear

what the equality sign is supposed to do; check whether two objects are equal or not. The idea is to give a list of all properties that the equality sign must fulfill. This is the theory of equality or *first-order logic with equality*, [DV01].

Definition 2.3.2 (First-Order Logic with Equality). The *interpreted predicate symbol* "=" with arity 2 is interpreted by the following axioms:

$$\forall x.x = x \qquad \text{(Reflexivity)}$$
$$\forall x \forall y.x = y \rightarrow y = x \qquad \text{(Symmetry)}$$
$$\forall x \forall y \forall z.(x = y \wedge y = z) \rightarrow x = z \qquad \text{(Transitivity)}$$

Additionally, for each functional symbol f with arity n and predicate symbol p with arity m, the *substitution axioms* hold

$$\forall x_1, ..., x_n \forall y_1, ..., y_n.(x_1 = y_1 \wedge ... \wedge x_n = y_n) \rightarrow f(x_1, ..., x_n) = f(y_1, ..., y_n) \qquad (2.7)$$
$$\forall x_1, ..., x_n \forall y_1, ..., y_n.(x_1 = y_1 \wedge ... \wedge x_n = y_n) \rightarrow (p(x_1, ..., x_m) \leftrightarrow p(y_1, ..., y_m)). \qquad (2.8)$$

For two given terms, it is also written $t_1 \neq t_2$ instead of $\neg(t_1 = t_2)$.

For the remaining part of this thesis we will always implicitly use the theory of equality, possibly extending it to more axioms that describe other symbols.

As a next step, the axioms described above are integrated into the sound and complete resolution calculus introduced in the previous section. This extension is called the Superposition Calculus, [NR01], the additional rules are stated in figure 2.3. The main idea of the superposition principle is being able to substitute terms directly. This is called the superposition rule. Its soundness can be derived from the substitution axioms for function and predicate symbols. It also includes equality resolution, stemming from the reflexivity axiom, and equality factoring, stemming from the symmetry and transitivity axiom. How these new rules are connected to the axioms for equality will become clearer in the proof of the next theorem.

Theorem 2.3.3 (Soundness and completeness of superposition calculus). *A first-order formula ϕ wrt. to the theory of equality is valid if and only if the empty clause $\square$ can be derived from $\neg\phi$ using only rules of Figures 2.2–2.3.*

Proof. We will prove soundness of the superposition, equality resolution, and equality factoring rule by using the sound binary resolution and factoring rules of Figure 2.2. We will assume that the terms s and s' are unifiable.

- Equality resolution can be derived using the reflexivity axiom. The following derivation holds

$$\frac{\underline{s \neq s' \vee C} \quad \underline{x = x}\ \text{Ax}}{C\theta}\ \text{BR}$$

  where $\theta = \mathsf{mgU}(s, s')$.

Superposition (Sup):

$$\frac{\underline{s = t} \vee C \quad L[s'] \vee D}{(L[t] \vee C \vee D)\theta}$$

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

Equality resolution (ER):

$$\frac{\underline{s \neq s'} \vee C}{C\theta}$$

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

Equality factoring (EF):

$$\frac{\underline{s = t} \vee \underline{s' = t'} \vee C}{(s = t \vee t \neq t' \vee C)\theta}$$

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

Figure 2.3: Additional rules of the superposition calculus $\mathbb{S}\mathrm{up}$, extending binary resolution and factoring from Figure 2.2. The expressions s, s', t, t' denote terms, L denotes a literal, C and D denote clauses.

- Equality factoring holds due to the transitivity axiom and the symmetry axiom (transformed into CNF):

$$\dfrac{\dfrac{s = t \vee \underline{s' = t'} \vee C \qquad \dfrac{\dfrac{}{x \neq y \vee \underline{y \neq z} \vee x = z}\,\mathrm{Ax} \quad \dfrac{}{\underline{y = z} \vee z \neq y}\,\mathrm{Ax}}{\underline{x \neq y} \vee x = z \vee z \neq y}\,\mathrm{BR}}{\underline{s = t} \vee C \vee \underline{s' = z} \vee z \neq t'}\,\mathrm{BR}}{(s = t \vee t \neq t' \vee C)\theta}\,\mathrm{F}$$

  where $\theta = \mathsf{mgU}(s, s')$. Note that in the last application of the binary resolution rule, the most general unifier of the literals $s' = t'$ and $x \neq y$ is used, $\theta' = \{x \mapsto s', y \mapsto t'\}$.

- The superposition rule holds due to the substitution axioms for predicate and functional symbols, that depending on how deep the term s is nested in literal L can be applied several times. Without loss of generality, we can assume that the term s' is always located at the first position of a predicate symbol p and function symbols $f_1, ..., f_n$. This leads to easier notation and does not change anything in the proof. We separate the proof into two cases.

  Case 1 (L is a positive literal).

  Let us firstly assume that $L[s']$ is positive literal, so a non-negated atomic formula of the form $p(f_n(...f_1(s', \overline{u}_1), \overline{u}_n), \overline{u}_p)$, where $\overline{u}_k = u_1^k, ..., u_{m_k}^k$ for $k \in \{1, ..., n, p\}$ are terms, $m_k + 1$ denotes the arity for functional symbol f_k, where $k \in \{1, ..., n\}$, and $m_p + 1$ denotes arity of predicate symbol p. We add " $+1$" for the terms in the first position. We know that the substitution axioms for functional symbols $f_1, ..., f_n$ hold,

(2.7), in particular

$$\begin{aligned} s = t \rightarrow f_1(s, \overline{u}_1) = f_1(t, \overline{u}_1) \\ f_1(s, \overline{u}_1) = f_1(t, \overline{u}_1) \rightarrow f_2(f_1(s, \overline{u}_1), \overline{u}_2) = f_2(f_1(t, \overline{u}_1), \overline{u}_2) \\ \ldots \\ f_{n-1}(\ldots f_1(s, \overline{u}_1) \ldots, \overline{u}_{n-1}) = f_{n-1}(\ldots f_1(t, \overline{u}_1) \ldots, \overline{u}_{n-1}) \rightarrow \\ f_n(\ldots f_1(s, \overline{u}_1) \ldots, \overline{u}_n) = f_n(\ldots f_1(t, \overline{u}_1) \ldots, \overline{u}_n). \end{aligned}$$

We can resolve these n axioms $n-1$ times into

$$s \neq t \vee f_n(\ldots f_1(s, \overline{u}_1) \ldots, \overline{u}_n) = f_n(\ldots f_1(t, \overline{u}_1) \ldots, \overline{u}_n). \tag{2.9}$$

We define the term $r[s] := f_n(\ldots f_1(s, \overline{u}_1) \ldots, \overline{u}_n)$ to shorten the notation. The weaker form (only forward implication) of the substitution axioms for predicate symbols, (2.8), holds:

$$r[s] = r[t] \rightarrow (p(r[s], \overline{u}_p) \rightarrow p(r[t], \overline{u}_p)). \tag{2.10}$$

We transform (2.10) into CNF which leads to

$$r[s] \neq r[t] \vee \neg p(r[s], \overline{u}_p) \vee p(r[t], \overline{u}_p). \tag{2.11}$$

Using (2.9) and (2.11), we can firstly derive the following using the left condition in the superposition rule of Figure 2.3.

$$\dfrac{\dfrac{\underline{s = t} \vee C \quad \overline{\underline{s \neq t} \vee r[s] = r[t]}\ \text{Ax}}{\underline{r[s] = r[t]} \vee C}\ \text{BR} \quad \dfrac{}{\underline{r[s] \neq r[t]} \vee \neg p(r[s], \overline{u}_p) \vee p(r[t], \overline{u}_p)}\ \text{Ax}}{\neg p(r[s], \overline{u}_p) \vee p(r[t], \overline{u}_p) \vee C}\ \text{BR}$$

Used together with the second condition of Figure 2.3, we have:

$$\dfrac{\underline{\neg p(r[s], \overline{u}_p)} \vee p(r[t], \overline{u}_p) \vee C \quad \underline{p(r[s'], \overline{u}_p)} \vee D}{(p(r[t], \overline{u}_p) \vee C \vee D)\theta}\ \text{BR}$$

where $\theta = \mathsf{mgU}(s, s')$.

Case 2 (L is a negative literal).

The second case is that $L[s']$ is a negative literal, so a negated atomic formula of the form $\neg p(f_n(\ldots f_1(s', \overline{u}_1), \overline{u}_n), \overline{u}_p)$. We use the same notation as before. The only difference is, that the backward implication of the substitution axiom for predicate symbols is used:

$$r[s] = r[t] \rightarrow (p(r[t], \overline{u}_p) \rightarrow p(r[s], \overline{u}_p)).$$

The derivation trees then look like

$$\dfrac{\dfrac{\underline{s = t} \vee C \quad \dfrac{}{\underline{s \neq t} \vee r[s] = r[t]}\,\text{Ax}}{r[s] = r[t] \vee C}\,\text{BR} \quad \dfrac{}{\underline{r[s] \neq r[t]} \vee \neg p(r[t], \overline{u}_p) \vee p(r[s], \overline{u}_p)}\,\text{Ax}}{\neg p(r[t], \overline{u}_p) \vee p(r[s], \overline{u}_p) \vee C}\,\text{BR}$$

and

$$\dfrac{\neg p(r[t], \overline{u}_p) \vee \underline{p(r[s], \overline{u}_p)} \vee C \quad \underline{\neg p(r[s'], \overline{u}_p)} \vee D}{(\neg p(r[t], \overline{u}_p) \vee C \vee D)\theta}\,\text{BR}$$

where $\theta = \mathsf{mgU}(s, s')$.

This concludes the soundness proof for the superposition rule.

We have established soundness of the included rules of Figure 2.3. The completeness proof goes analogously as the proof of (2.6). □

A thorough example of how the superposition calculus works in theory will be presented in Chapter 3. At a later stage, when the recursive synthesis framework is introduced, we will also see how an automated proof (with synthesis) looks like in practice, using the first-order theorem prover Vampire, see Section 5.3.

3 First Revisit of Motivating Example

In this chapter, we translate the proof from Section 1.1 into a formal one, using the superposition calculus.

Let us restate the specification, whose validity we want to establish:

$$\forall x_1, x_2, x_3 \in \mathbb{N}.\ \exists y \in \mathbb{N}.\ x_1 \cdot x_2 + x_1 \cdot x_3 = x_1 \cdot y. \tag{SPD}$$

Validity in this case means validity with respect to an underlying theory, which is the theory of equality with additional axioms that define the natural numbers $\mathbb{N}$. These additional axioms are given in Figure 3.1. Hence, all of these axioms can be used to prove validity of (SPD).

Note here that for now, we are not yet interested in synthesizing a function that fulfills specification (SPD), but only to derive a proof establishing validity of it. For this, the first step is preprocessing the formula into CNF, as described in Section 2.2.

Preprocessing. Firstly, the formula (SPD) is negated, which entails flipping all quantifiers and negating the equation, leading to

$$\exists x_1, x_2, x_3 \in \mathbb{N}.\ \forall y \in \mathbb{N}.\ x_1 \cdot x_2 + x_1 \cdot x_3 \neq x_1 \cdot y. \tag{SPD$_{neg}$}$$

The formula only consists of one literal and therefore all quantifiers are already at the outermost position. Consequently, only skolemization is needed to transform (SPD$_{neg}$) into CNF. Because there is no universal quantifier coming before the three existential quantifiers, we transform the variables x_1, x_2 and x_3 into skolem constants that we will denote with σ_1, σ_2 and σ_3. After this is done, the formula looks like

$$\forall y \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_3 \neq \sigma_1 \cdot y. \tag{SPD$_{cnf}$}$$

To save space, the universal quantifier is removed, which leads to the final input,

$$\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_3 \neq \sigma_1 \cdot y. \tag{SPD$_{in}$}$$

Proof derivation. When formalizing the proof from Section 1.1 using the superposition calculus, one possible derivation is the following.

1. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_3 \neq \sigma_1 \cdot y$ [(SPD$_{in}$)]
2. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n = \sigma_1 \cdot \sigma_v \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot f(x)$ [(IndAx1)]
3. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) \neq \sigma_1 \cdot u_{\mathsf{s}} \vee$
   $\vee\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot f(x)$ [(IndAx2)]
4. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n = \sigma_1 \cdot \sigma_v$ [BR 1, 2]

$\forall x.\ \mathsf{s}(x) \neq 0$	(Nat1)
$\forall x, y.\ \mathsf{s}(x) = \mathsf{s}(y) \to x = y$	(Nat2)
$\forall x \in \mathbb{N}. x + 0 = x$	(Add1)
$\forall x, n \in \mathbb{N}. x + \mathsf{s}(n) = \mathsf{s}(x + n)$	(Add2)
$\forall x \in \mathbb{N}. x \cdot 0 = 0$	(Mult1)
$\forall x, n \in \mathbb{N}. x \cdot \mathsf{s}(n) = x \cdot n + x$	(Mult2)
$(F[0] \wedge \forall n.(F[n] \to F[\mathsf{s}(n)])) \to \forall x. F[x].$	(IndAx)

Figure 3.1: Some of the axioms in the theory of natural numbers $\mathbb{N}$.

5. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) \neq \sigma_1 \cdot u_{\mathsf{s}}$ [BR 1, 3]
6. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n + \sigma_1 \neq \sigma_1 \cdot u_{\mathsf{s}}$ [Sup 5, (Mult2)]
7. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_v + \sigma_1 \neq \sigma_1 \cdot u_{\mathsf{s}}$ [Sup 4, 6]
8. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_v + \sigma_1 \neq \sigma_1 \cdot u_{\mathsf{s}}$ [F 7]
9. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \mathsf{s}(\sigma_v) \neq \sigma_1 \cdot u_{\mathsf{s}}$ [Sup 8, (Mult2)]
10. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0$ [ER 9]
11. $\sigma_1 \cdot \sigma_2 + 0 \neq \sigma_1 \cdot u_0$ [Sup 10, (Mult1)]
12. $\sigma_1 \cdot \sigma_2 \neq \sigma_1 \cdot u_0$ [Sup 11, (Add1)]
13. $\square$ [ER 12]

Going through the formal proof. Let us take a closer look at the proof derivation. The first observation is, that the last line consists only of the empty clause $\square$, which means that the derivation was indeed successful, validity of (SPD) has been proven. We will now go through all the steps leading to the empty clause. The inference rules that were used can be found in Figures 2.2–2.3.

Preprocessing from above gives us the input clause 1. So our clause set consists of a clause containing one literal. To derive the empty clause, we need to construct a literal of the form $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_3 = \sigma_1 \cdot x$. This is done by using the induction axiom (IndAx) that holds for any arbitrary formula F.

We use the formula $F[x] : \exists\, y.\, \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y$, so when plugging formula F in the induction axiom we get

$$\begin{aligned}&((\exists y \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot y) \wedge \\ \forall n \in \mathbb{N}.((\exists y \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot n = \sigma_1 \cdot y) &\rightarrow (\exists y \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(n) = \sigma_1 \cdot y)) \\ &\rightarrow \forall x \in \mathbb{N}.(\exists y \in \mathbb{N}.\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y)).\end{aligned} \tag{IndAx'}$$

The induction axiom, as the name states, is an axiom in the theory $\mathbb{N}$ of natural numbers. Therefore, we can add it to our clause set. When doing this we first have to transform (IndAx') into CNF. We go through all the steps described in Section 2.2.1. This includes cleansing, where we rename the first occurrence of variable y to u_0, symbolizing the base case, and the second occurrence of variable y to u_s, symbolizing the step case. This leads to

$$\begin{aligned}&(\exists u_0 \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge \\ \forall n \in \mathbb{N}.(\exists v \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot n = \sigma_1 \cdot v &\rightarrow \exists u_\mathsf{s} \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(n) = \sigma_1 \cdot u_\mathsf{s})) \\ &\rightarrow \forall x \in \mathbb{N}.(\exists y \in \mathbb{N}.\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y).\end{aligned} \tag{IndAxClean}$$

The transformation into PNF, as stated before is not unique. We choose the PNF formula where the fewest dependencies take place:

$$\begin{aligned}&\exists n \in \mathbb{N}\ \exists v \in \mathbb{N}\ \forall x \in \mathbb{N}\ \exists y \in \mathbb{N}\ \forall u_0 \in \mathbb{N}\ \forall u_\mathsf{s} \in \mathbb{N}. \\ &((\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge \\ &(\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot n = \sigma_1 \cdot v \rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(n) = \sigma_1 \cdot u_\mathsf{s})) \\ &\rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y).\end{aligned} \tag{IndAxPNF}$$

Now, we skolemize the existentially quantified variables, σ_v is the skolem constant for variable v, σ_n the skolem constant for variable n, and f the skolem function of arity 1 for variable y:

$$\begin{aligned}&\forall x \in \mathbb{N}\ \forall u_0 \in \mathbb{N}\ \forall u_\mathsf{s} \in \mathbb{N}. \\ &(\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge \\ &\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n = \sigma_1 \cdot \sigma_v \rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) = \sigma_1 \cdot u_\mathsf{s})) \\ &\rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot f(x)).\end{aligned} \tag{IndAxSko}$$

As a last step, the quantifier-free segment is transformed into CNF, resulting in two clauses.

$$\begin{aligned}&\forall x \in \mathbb{N}\ \forall u_0 \in \mathbb{N}\ \forall u_\mathsf{s} \in \mathbb{N}. \\ &(\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n = \sigma_1 \cdot \sigma_v \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot f(x))\end{aligned} \tag{IndAx1}$$

$$\wedge(\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) \neq \sigma_1 \cdot u_\mathsf{s} \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot f(x)) \tag{IndAx2}$$

The two clauses, (IndAx1) and (IndAx2), can then be added to our clause set and used for resolving the literal $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_3 \neq \sigma_1 \cdot y$ in clause 1 with $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot f(x)$, using

the substitution $\theta_1 : \{x \mapsto \sigma_3, y \mapsto f(\sigma_3)\}$. This results in clauses 4 and 5 respectively. Our goal is to change the literal $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) \not\approx \sigma_1 \cdot u_{\mathsf{s}}$ in clause 5 in such a way that the term $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n$ appears and can therefore be substituted, using the literal in clause 4, with $\sigma_1 \cdot \sigma_v$. The idea behind this is to use the induction hypothesis, which is exactly the second literal in clause 4, to be able to resolve the negated induction assumption, the second literal in clause 5. This is where the superposition calculus comes into play. We state the (Mult2) axiom again

$$\forall x, n \in \mathbb{N}. x \cdot \mathsf{s}(n) = x \cdot n + x. \tag{Mult2}$$

We apply the substitution $\theta_2 : \{x \mapsto \sigma_1, n \mapsto \sigma_n\}$ on (Mult2) resulting in

$$\sigma_1 \cdot \mathsf{s}(\sigma_n) = \sigma_1 \cdot \sigma_n + \sigma_1, \tag{3.1}$$

and see that the term on the right side of the equality in (3.1) is contained in the second literal in clause 5. So using θ_2, we can apply the superposition rule on the clause (Mult2) and clause 5 in our derivation, where the underlined term is $\sigma_1 \cdot \mathsf{s}(\sigma_n)$, resulting in clause 6.

Now the changed literal in clause 6 contains the term $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n$ and this term also appears in the second literal in clause 4. Therefore, we can apply the superposition rule again on clauses 4 and 6, this time not even needing a substitution (or using the identity substitution, to be exact).

In the derived clause 7, the literal $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \not\approx \sigma_1 \cdot u_0$ appears twice but can be removed with the factoring rule, resulting in clause 8.

We apply superposition once again with the axiom (Mult2) resulting in clause 9. Now, the second literal can be easily resolved using equality resolution with the substitution $\theta_3 : \{u_{\mathsf{s}} \mapsto \mathsf{s}(\sigma_v)\}$.

What is left to do is to simplify the term on the right side of the equality until one can resolve the last literal in clause 10. This is done with the axioms (Mult1) and (Add1) resulting in clauses 11 and 12 respectively. As a last step, equality resolution is applied using the substitution $\theta_4 : \{u_0 \mapsto \sigma_2\}$, resulting in the empty clause, which concludes the formal proof.

4 Synthesis of Non-recursive Programs in Saturation

The findings that are used for this chapter have been developed by Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov in [HKNV23]. The notions, definitions, and properties of this section are thus based upon [HKNV23].

Following and recalling results of [HKNV23], in Section 4.1 we introduce the notions of answer literals and show how to use answer literals to track changes in the proof derivation. Answer literals are used to adapt the superposition calculus in such a way that synthesis of non-recursive programs is possible.

In Section 4.2 we formulate a small problem to showcase what the introduced framework can do.

In Section 4.3 we explain what the limitations of the framework are regarding recursive programs.

4.1 Tracking Changes in Proof Derivations with Answer Literals

Firstly the definition of a synthesis specification is stated, as a formula written in first-order logic. This formula, the specification, is supposed to convey the intended behavior of the program that should be synthesized.

Definition 4.1.1 (Synthesis Specification). Let $\overline{x} = x_1, ..., x_n$ and y be the only free variables of an arbitrary formula F in first-order logic with equality. Then we call

$$\forall \overline{x} \, \exists y. F[\overline{x}, y] \tag{4.1}$$

a *synthesis specification* with input $\overline{x}$ and output y. We call a term r a *witness* of the synthesis specification (4.1) for the variable y if the formula

$$\forall \overline{x}. F[\overline{x}, r[\overline{x}]] \tag{4.2}$$

is valid.

The *synthesis task* is then to prove validity of a given specification of the form (4.1) and to simultaneously compute a witness for which the specification holds. *Answer literals* are used to track changes in the proof and will be denoted with $\mathsf{ans}(.)$.

Answer literals. When looking back at the motivating example of Section 1.1 we have seen that while resolving literals, the algorithm applied substitutions on different variables. We have also seen that these substitutions translated exactly to what the task of our program was. The idea of answer literal is to track the changes of the output variable y by

storing the substitutions that are applied on y, written as $\mathsf{ans}(y)$, [Gre69]. Answer literals do not change anything in the proof and are also not used for deriving new clauses. If at some point a clause is derived that contains only an answer literal, this is the pendant to the usual proving process, the empty clause, which means that a refutation has been found. The term inside the answer literal of the last step is then used to construct a witness for the given specification.

Preprocessing of synthesis specification. The proof principle stays the same, namely proving by refutation. Therefore, the input specification (4.1) is negated and nextly transformed into CNF, see Section 2.2.1. This results in a first-order logic formula of the form

$$\mathsf{cnf}(\exists\, \overline{x}\, \forall y. \neg F[\overline{x}, y]). \tag{4.3}$$

The distinct part of preprocessing for synthesis is to add to each clause in (4.3) an answer literal that is dependent on the output of our specification, i.e. $\mathsf{ans}(y)$. This then leads to the following input

$$\mathsf{cnf}(\mathsf{cnf}(\exists\, \overline{x}\, \forall y. \neg F[\overline{x}, y]) \vee \mathsf{ans}(y)). \tag{4.4}$$

The resulting set of clauses is the final input, which concludes preprocessing.

Introducing Superposition Calculus with Synthesis. The inference rules of the superposition calculus for synthesis are stated in Figure 4.1. One thing to ensure is that the answer literals are not used for the proof. For this, the inference rules from the superposition calculus are adapted. For discussing these new rules in more detail the following concept is introduced.

Definition 4.1.2 (Abstract Unifier). We call a pair (θ, D) an *abstract unifier* of two expressions E_1 and E_2 if

1. θ is a substitution and D a disjunction of disequalities, i.e. $D \simeq (s_1 \not\simeq t_1 \vee ... \vee s_n \not\simeq t_n)$, where $s_1, ..., s_n$ and $t_1, ..., t_n$ are terms and
2. $(D \vee E_1 \simeq E_2)\theta$ is valid in the underlying theory.

Remark 4.1.3 To clarify the intended meaning of an abstract unifier we rewrite the second bullet point of Definition 4.1.2 into

$$(s_1 = t_1 \wedge ... \wedge s_n = t_n)\theta \rightarrow (E_1 \simeq E_2)\theta.$$

We see that the substitution of the abstract unifier only surely unifies the expressions E_1 and E_2 if all the equalities in D with respect to substitution θ are fulfilled. Essentially, an abstract unifier is a most general unifier with conditions. On the other hand, every most general unifier is an abstract unifier where D is the empty set $\emptyset$.

We state a simple example of an abstract unifier.

Example 4.1.4 Let us look at the atomic formulas $A_1 : f(x_1, c) = g(x_2)$ and $A_2 : f(g(x_3), x_4) = x_5$. For given condition $x_1 = g(x_3)$, the substitution $\theta : \{x_4 \mapsto c, x_5 \mapsto g(x_2)\}$ is a unifier of A_1 and A_2. Therefore, the tuple $(\theta, \{x_1 \neq g(x_3)\})$ is an abstract unifier of A_1 and A_2.

Definition 4.1.5 (Computable Symbol). We differentiate between *computable* and *uncomputable symbols*. A symbol is computable if there is a program that can evaluate it. We say that an expression E is *computable*, if it only contains computable symbols.

We want our synthesized program to only consist of computable expression, therefore we further adapt the definition of abstract unifiers to the following.

Definition 4.1.6 (Computable Unifier). We call (θ, D) a *computable unifier* of two expressions E_1 and E_2 with respect to expression E_3 if (θ, D) is an abstract unifier of E_1 and E_2 and the expression $E_3\theta$ is computable.

Remark 4.1.7 The concept of computable unifiers ensures that an expression is still computable after the substitution has been applied. Therefore, by restricting the unifiers to be computable, only computable expressions will appear inside the answer literal, see Figure 4.1.

Example 4.1.8 For any expression E_3, the abstract unifier (θ, D) from Example 4.1.4 is a computable one, as long as the expression $E_3\theta$ is computable.

To be able to differentiate between different cases we introduce the following notion.

Definition 4.1.9 (if − then − else−Constructor). For arbitrary terms s, t and an arbitrary atomic formula A we define

$$\text{if } A \text{ then } s \text{ else } t := \begin{cases} s, & \text{if } A \text{ is valid} \\ t, & \text{else.} \end{cases}$$

Saturation process with synthesis. In Figure 4.1 the adapted rules of the superposition calculus are stated and will be discussed in the following.

1. We start with the simplest modification first: Factoring, equality resolution, and equality factoring. For these rules, only one clause is in the precondition. Therefore, the answer literal is treated like the other remaining literals in the clause and not modified, except for the application of the substitution. Due to the fact that we want to avoid uncomputable symbols occurring inside the answer literal, we are only considering computable unifiers with respect to the term r that is inside the answer literal.

2. For binary resolution there are two possibilities. First, the terms inside the answer literals, r, and r' can be unified. If that is possible, we first apply binary resolution (note here that the term $r \neq r'$ is added to the postcondition) and then equality

Binary resolution with condition (BR'):

$$\frac{\underline{L} \vee C \vee \mathsf{ans}(r) \quad \underline{\neg L'} \vee C' \vee \mathsf{ans}(r')}{(D \vee C \vee C' \vee \mathsf{ans}(\mathsf{if}\ L\ \mathsf{then}\ r'\ \mathsf{else}\ r))\theta}$$

where (θ, D) is a computable unifier of L, L' wrt. $\mathsf{if}\ L\ \mathsf{then}\ r'\ \mathsf{else}\ r$

Binary resolution (BR):

$$\frac{\underline{L} \vee C \vee \mathsf{ans}(r) \quad \underline{\neg L'} \vee C' \vee \mathsf{ans}(r')}{(D \vee C \vee C' \vee r \neq r' \vee \mathsf{ans}(r))\theta}$$

where (θ, D) is a computable unifier of L, L' wrt. r.

Superposition with condition (Sup'):

$$\frac{\underline{s = t} \vee C \vee \mathsf{ans}(r) \quad \underline{L[s']} \vee C' \vee \mathsf{ans}(r')}{(D \vee L[t] \vee C \vee C' \vee \mathsf{ans}(\mathsf{if}\ s = t\ \mathsf{then}\ r'\ \mathsf{else}\ r))\theta}$$

where (θ, D) is a computable unifier of s, s' wrt. $\mathsf{if}\ s = t\ \mathsf{then}\ r'\ \mathsf{else}\ r$.

Superposition (Sup):

$$\frac{\underline{s = t} \vee C \vee \mathsf{ans}(r) \quad \underline{L[s']} \vee D \vee \mathsf{ans}(r')}{(D \vee L[t] \vee C \vee C' \vee r \neq r' \vee \mathsf{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, s' wrt. r.

Factoring (F):

$$\frac{\underline{L} \vee \underline{L'} \vee C \vee \mathsf{ans}(r)}{(D \vee L \vee C \vee \mathsf{ans}(r))\theta}$$

where (θ, D) is a computable unifier of L, L' wrt. r.

Equality resolution (ER):

$$\frac{\underline{s \neq s'} \vee C \vee \mathsf{ans}(r)}{(D \vee C \vee \mathsf{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, s' wrt. r.

Equality factoring (EF):

$$\frac{\underline{s = t} \vee \underline{s' = t'} \vee C \vee \mathsf{ans}(r)}{(D \vee s = t \vee t \neq t' \vee C \vee \mathsf{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, s' wrt. r.

Figure 4.1: Adapted rules of the superposition calculus $\mathbb{S}\mathrm{up}$ including answer literals for non-recursive synthesis. The expressions s, s', t, t' denote terms, L denotes a literal, C and D denote clauses.

resolution. Second, if the terms r and r' are in fact not unifiable, we need to introduce the $\mathsf{if} - \mathsf{then} - \mathsf{else}$ constructor. Binary resolution is applied in the usual way and the term inside the answer literal changes to $\mathsf{if}\ L\ \mathsf{then}\ r'\ \mathsf{else}\ r$. We call this adapted rule *binary resolution with condition*.

3. The modification of the superposition rule works in a similar way to binary resolution. In the first case when terms r and r' are unifiable, we apply superposition and equality

resolution. The second case is that r and r' are not unifiable; then the term inside the answer literal changes to $\mathsf{if}\ s = t\ \mathsf{then}\ r'\ \mathsf{else}\ r$. We call this adapted rule *superposition with condition*.

The following theorem ensures that the adapted rules in Figure 4.1 are sound.

Theorem 4.1.10 (Correctness of Superposition with Synthesis). *The inference rules of the adapted superposition calculus $\mathbb{S}up$ with answer literals, stated in Figure 4.1, are sound.*
Proof. Lemma 4 in Appendix A in [HKNV23]. □

The next theorem states that if a refutation using the adapted rules in Figure 4.1 has been found, this also entails the detection of a witness for the input specification.

Theorem 4.1.11 *Assume that for a specification of the form (4.1) a clause containing only the answer literal has been derived using only rules in Figure 4.1. Then the term inside the answer literal is a witness of specification (4.1).*
Proof. Theorem 1 in Appendix A in [HKNV23]. □

4.2 Illustrative Example: Maximum of Two Naturals

We show a small example to showcase the utility of described superposition calculus with answer literals. The synthesis task is to come up with a program that computes the maximum of two given natural numbers. For stating the specification we need the predicate symbol $\leq$ of arity 2 that determines if one natural is smaller or equal than the other.

Definition 4.2.1 (Ordering on Naturals). We define a predicate symbol "$\leq$" of arity 2 inductively.

$$\forall x.(x \leq 0 \leftrightarrow x = 0) \tag{Order1}$$
$$\forall x, y.(x \leq \mathsf{s}(y) \leftrightarrow (x \leq y \vee x = \mathsf{s}(y))) \tag{Order2}$$

Lemma 4.2.2 *The predicate symbol "$\leq$" defined by (Order1) and (Order2) has the following property:*

$$\forall x.x \leq x. \tag{Reflexivity}$$

Proof. The ordering "$\leq$" was defined inductively, therefore it comes intuitively that we also prove by induction. PLugging in the induction axiom with the formula $F[x] : x \leq x$ leads to

$$(0 \leq 0 \wedge \forall n.(n \leq n \rightarrow \mathsf{s}(n) \leq \mathsf{s}(n))) \rightarrow \forall x.x \leq x. \tag{4.5}$$

We transform (4.5) into CNF, resulting in the two clauses

$$\neg 0 \leq 0 \vee \sigma_n \leq \sigma_n \vee x \leq x \tag{4.6}$$
$$\neg 0 \leq 0 \vee \neg \mathsf{s}(\sigma_n) \leq \mathsf{s}(\sigma_n) \vee x \leq x. \tag{4.7}$$

We use the second clause, (4.7), together with the following two clauses resulting from transforming the ordering axioms into CNF,

$$\forall x.x \neq 0 \vee x \leq 0 \tag{4.8}$$
$$\forall x\, \forall y.x \neq \mathsf{s}(y) \vee x \leq \mathsf{s}(y), \tag{4.9}$$

to derive the following formal proof.

1. $\neg(\sigma_x \leq \sigma_x)$ [input]
2. $\neg 0 \leq 0 \vee \neg \mathsf{s}(\sigma_n) \leq \mathsf{s}(\sigma_n) \vee x \leq x$ [(4.7)]
3. $\neg 0 \leq 0 \vee \neg \mathsf{s}(\sigma_n) \leq \mathsf{s}(\sigma_n)$ [BR 1, 2]
4. $x \neq 0 \vee x \leq 0$ [(4.8)]
5. $z = z$ [(Reflexivity)]
6. $0 \leq 0$ [BR 4, 5]
7. $\neg \mathsf{s}(\sigma_n) \leq \mathsf{s}(\sigma_n)$ [BR 3, 6]
8. $x \neq \mathsf{s}(y) \vee x \leq \mathsf{s}(y)$ [(4.9)]
9. $\mathsf{s}(\sigma_n) \leq \mathsf{s}(\sigma_n)$ [BR 5, 8]
10. $\square$ [BR 7, 9]

We used the following substitutions: $\theta_1 : \{x \mapsto \sigma_x\}$ for resolving clause 1 and 2, $\theta_2 : \{x \mapsto 0, z \mapsto 0\}$ for resolving clause 4 and 5 and finally $\theta_3 : \{x \mapsto \mathsf{s}(\sigma_n), y \mapsto \sigma_n, z \mapsto \mathsf{s}(\sigma_n)\}$ for resolving clause 5 and 8. $\square$

Using the reflexivity property of "$\leq$", we can prove the following specification

$$\forall x_1, x_2 \in \mathbb{N}.\exists y \in \mathbb{N}.x_1 \leq y \wedge x_2 \leq y \wedge (x_1 = y \vee x_2 = y). \tag{SpecMaxNat}$$

For preprocessing, the specification (SpecMaxNat) is negated and transformed into CNF; this entails flipping the quantifiers, skolemizing the variables x_1 and x_2 and multiplying out the two clauses. In addition to the usual preprocessing, we add the answer literal $\mathsf{ans}(y)$ to both of the clauses, leading to

$$\neg(\sigma_1 \leq y) \vee \neg(\sigma_2 \leq y) \vee \sigma_1 \neq y \vee \mathsf{ans}(y) \tag{C1}$$
$$\neg(\sigma_1 \leq y) \vee \neg(\sigma_2 \leq y) \vee \sigma_2 \neq y \vee \mathsf{ans}(y). \tag{C2}$$

This is then the input set of clauses for superposition with synthesis. We can come up with the following derivation.

1. $\neg(\sigma_1 \leq y) \vee \neg(\sigma_2 \leq y) \vee \sigma_1 \neq y \vee \mathsf{ans}(y)$ [(C1)]

2. $\neg(\sigma_1 \leq y) \vee \neg(\sigma_2 \leq y) \vee \sigma_2 \neq y \vee \mathsf{ans}(y)$ [(C2)]
3. $\neg(\sigma_1 \leq \sigma_1) \vee \neg(\sigma_2 \leq \sigma_1) \vee \mathsf{ans}(\sigma_1)$ [ER 1]
4. $\neg(\sigma_1 \leq \sigma_2) \vee \neg(\sigma_2 \leq \sigma_2) \vee \mathsf{ans}(\sigma_2)$ [ER 2]
5. $\neg(\sigma_2 \leq \sigma_1) \vee \mathsf{ans}(\sigma_1)$ [BR 3, (Reflexivity)]
6. $\neg(\sigma_1 \leq \sigma_2) \vee \mathsf{ans}(\sigma_2)$ [BR 4, (Reflexivity)]
7. $\mathsf{ans}(\mathsf{if}\ \sigma_1 \leq \sigma_2\ \mathsf{then}\ \sigma_2\ \mathsf{else}\ \sigma_1)$ [BR' 5, 6]
8. $\square$ [answer literal removal]

Going through program derivation. We apply equality resolution on both clause (C1) ($\theta_1 : \{y \mapsto \sigma_1\}$) and clause (C2) ($\theta_2 : \{y \mapsto \sigma_2\}$), resulting in clauses 3 and 4. Note here that this resolution results in a change of the answer literal. We can resolve the literal $\neg(\sigma_1 \leq \sigma_1)$ in clause 3, and $\neg(\sigma_2 \leq \sigma_2)$ in clause 4, by using the reflexivity axiom, resulting in clause 5 and clause 6, respectively. Note that the skolem constants σ_1 and σ_2 that appear inside the answer literals are not unifiable. Therefore, the binary resolution rule with condition is used to obtain clause 7. As a last step, the answer literal is removed resulting in the empty clause. The proving process then terminates and returns the answer literal. Postprocessing consists of mapping the skolem constants back to their original variable. The output is therefore the following program

$$\mathsf{if}\ x_1 \leq x_2\ \mathsf{then}\ x_2\ \mathsf{else}\ x_1,$$

which can easily be verified as the program that computes the maximum of two numbers. Interestingly enough this example could also be synthesized in a recursive way due to the definition of the ordering.

4.3 Limitations

We will shortly reference back to Chapter 3 and state the program derivation used to prove validity of the law of distributivity using the basic superposition calculus.

1. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_3 \not\approx \sigma_1 \cdot y$ [input]
2. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n = \sigma_1 \cdot \sigma_v \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot \boldsymbol{f(x)}$ [(IndAx1)]
3. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) \not\approx \sigma_1 \cdot u_\mathsf{s} \vee$
   $\vee\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot \boldsymbol{f(x)}$ [(IndAx2)]
4. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n = \sigma_1 \cdot \sigma_v$ [BR 1, 2]
5. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) \not\approx \sigma_1 \cdot u_\mathsf{s}$ [BR 1, 3]
6. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n + \sigma_1 \not\approx \sigma_1 \cdot u_\mathsf{s}$ [Sup 5, (Mult2)]
7. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_v + \sigma_1 \not\approx \sigma_1 \cdot u_\mathsf{s}$ [Sup 4, 6]

8. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_v + \sigma_1 \not\approx \sigma_1 \cdot u_{\mathsf{s}}$ [F 7]
9. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \mathsf{s}(\sigma_v) \not\approx \sigma_1 \cdot u_{\mathsf{s}}$ [Sup 8, (Mult2)]
10. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \not\approx \sigma_1 \cdot u_0$ [ER 9]
11. $\sigma_1 \cdot \sigma_2 + 0 \not\approx \sigma_1 \cdot u_0$ [Sup 10, (Mult1)]
12. $\sigma_1 \cdot \sigma_2 \not\approx \sigma_1 \cdot u_0$ [Sup 11, (Add1)]
13. $\square$ [ER 12]

Let us look at this derivation in more detail and explain the issues with the described synthesis framework from Section 4.1 in regards to this specific example.

When the induction axiom was preprocessed, the skolem function f was introduced. The function symbol f (highlighted in bold) therefore appears in clauses 2 and 3. In the usual superposition proof, the expressions $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_3 \not\approx \sigma_1 \cdot y$ and $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x \approx \sigma_1 \cdot f(x)$ are resolved together with the substitution $\theta : \{y \mapsto f(x)\}$. However, this step cannot be done with the introduced synthesis framework, due to the fact that f is not interpreted and hence, also not computable. Therefore the expression $f(x)$ is restricted from being substituted into the answer literal and the clauses in lines 2 and 3 cannot be resolved further. This shows that the use of the induction axiom together with this synthesis framework needs to be adapted further. We will see how this issue is resolved in Chapter 5.

5 Second Revisit of Motivating Example

We now explain how the synthesis framework described in Chapter 4 can be further extended to also being able to synthesize recursive functions, see Section 5.1. We will showcase this with the motivating example that was already formally proven in Chapter 3. After giving the theoretical proof derivation with synthesis in Section 5.2, we will also show how this synthesis task can look in practice, using the first-order theorem prover Vampire, see Section 5.3.

5.1 From Non-recursive to Recursive Synthesis: Changing Order of Quantifiers

In the following section we explain how recursive synthesis is possible based on the motivating example introduced in Section 1.1.

The goal is to synthesize a function that fulfills the following specification:

$$\forall x_1, x_2, x_3 \in \mathbb{N}.\ \exists y \in \mathbb{N}.\ x_1 \cdot x_2 + x_1 \cdot x_3 = x_1 \cdot y. \tag{SPD}$$

We look in detail how the induction axiom was preprocessed in Chapter 3 before adding it to the set of clauses used for deriving the empty clause.

We have used the induction axiom

$$(F[0] \wedge \forall n.(F[n] \rightarrow F[\mathsf{s}(n)])) \rightarrow \forall x.F[x] \tag{IndAx}$$

with the first-order logic formula

$$F[x] : \exists\, y.\, \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y. \tag{5.1}$$

After inserting the specified formula (5.1) and cleansing (IndAx) we get

$$\begin{gathered}(\exists u_0 \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge \\ \forall n \in \mathbb{N}.(\exists w \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot n = \sigma_1 \cdot w \rightarrow \exists u_{\mathsf{s}} \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(n) = \sigma_1 \cdot u_{\mathsf{s}})) \\ \rightarrow \forall x \in \mathbb{N}.(\exists y \in \mathbb{N}.\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y).\end{gathered} \tag{IndAxClean}$$

As a next step the induction axiom was transformed into PNF, resulting in the following formula:

$$\begin{gathered}\exists n \in \mathbb{N}\ \exists w \in \mathbb{N}\ \forall \boldsymbol{x} \in \mathbb{N} \exists \boldsymbol{y} \in \mathbb{N} \forall \boldsymbol{u_0} \in \mathbb{N} \forall \boldsymbol{u_{\mathsf{s}}} \in \mathbb{N}. \\ ((\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge \\ (\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot n = \sigma_1 \cdot w \rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(n) = \sigma_1 \cdot u_{\mathsf{s}})) \\ \rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y).\end{gathered} \tag{IndAxPNFOld}$$

We want to emphasize on the order of the quantifiers in (IndAxPNFOld). In particular, we see that the existential quantifier of variable y is coming before the universal quantifiers of variables u_0 and u_s. When skolemizing (IndAxPNFOld) this therefore results in

$$\begin{aligned}&\forall x \in \mathbb{N} \forall u_0 \in \mathbb{N} \forall u_\mathsf{s} \in \mathbb{N}.\\ &(\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge\\ &\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n = \sigma_1 \cdot \sigma_w \rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) = \sigma_1 \cdot u_\mathsf{s}))\\ &\rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot \boldsymbol{f(x)}).\end{aligned} \tag{IndAxSkoOld}$$

We see in (IndAxSkoOld) that the skolem function f of variable y only depends on variable x. For just proving the specification (SPD), as was done in Chapter 3, this order of the quantifier suffices. But for synthesizing a program we also want to track changes in the answer literal of variables u_0 and u_s and construct a recursive function with their stored values. Therefore, we want to change the order of the quantifiers in such a way that the universal quantifiers of variables u_0 and u_s come *before* the existential quantifier of variable y. Then, during skolemizing, we will get a *computable* function, denoted with rec, that depends not only on the input x but also on the base case variable (u_0) and the step case variable (u_s).

An important thing to note here is that there are different transformations into PNF that are all logically equivalent to the original formula.

Let us look at this in more detail. Starting from the cleansed formula of the induction axiom,

$$\begin{aligned}&(\exists u_0 \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge\\ &\forall n \in \mathbb{N}.(\exists w \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot n = \sigma_1 \cdot w \rightarrow \exists u_\mathsf{s} \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(n) = \sigma_1 \cdot u_\mathsf{s}))\\ &\rightarrow \forall x \in \mathbb{N}.(\exists y \in \mathbb{N}.\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y),\end{aligned} \tag{IndAxClean}$$

we firstly pull out the quantifiers of variables w and u_s resulting in

$$\begin{aligned}&(\exists u_0 \in \mathbb{N}.\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge\\ &\forall n \in \mathbb{N} \forall w \in \mathbb{N} \exists u_\mathsf{s} \in \mathbb{N}.(\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot n = \sigma_1 \cdot w \rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(n) = \sigma_1 \cdot u_\mathsf{s}))\\ &\rightarrow \forall x \in \mathbb{N}.(\exists y \in \mathbb{N}.\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y).\end{aligned} \tag{IndAx'}$$

Nextly, the quantifiers are pulled out of the outermost premise in the following order

$$\begin{aligned}&\exists n \in \mathbb{N}\ \exists w \in \mathbb{N}\ \forall u_0 \in \mathbb{N}\ \forall u_\mathsf{s} \in \mathbb{N}.\ (\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge\\ &(\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot n = \sigma_1 \cdot w \rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(n) = \sigma_1 \cdot u_\mathsf{s}))\\ &\rightarrow \forall x \in \mathbb{N}.(\exists y \in \mathbb{N}.\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y).\end{aligned} \tag{IndAx''}$$

Only then the quantifiers of variable x and y are pulled out, with the final result of

$$\begin{aligned}&\exists n \in \mathbb{N}\, \exists w \in \mathbb{N}\, \forall u_0 \in \mathbb{N}\, \forall u_\mathsf{s} \in \mathbb{N}\, \forall x \in \mathbb{N}\, \exists y \in \mathbb{N}.\\ &((\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge\\ &\ (\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot n = \sigma_1 \cdot w \rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(n) = \sigma_1 \cdot u_\mathsf{s}))\\ &\rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y).\end{aligned} \tag{IndAxPNFNew}$$

We firstly skolemize the bounded variables n and w in (IndAxPNFNew), leading to

$$\begin{aligned}&\forall u_0 \in \mathbb{N} \forall u_\mathsf{s} \in \mathbb{N} \forall x \in \mathbb{N} \exists y \in \mathbb{N}.\ (\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge\\ &(\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n = \sigma_1 \cdot \sigma_w \rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) = \sigma_1 \cdot u_\mathsf{s}))\\ &\qquad \rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot y,\end{aligned} \tag{IndAxSkoNew'}$$

where σ_n is a skolem constant of n and σ_w is a skolem constant of w.

Now one can see in (IndAxSkoNew') that when variable y is skolemized, a skolem function with three inputs u_0, u_s and x will be introduced. We will denote this skolem function with rec. This results in the following formula:

$$\begin{aligned}&\forall u_0 \in \mathbb{N} \forall u_\mathsf{s} \in \mathbb{N} \forall x \in \mathbb{N}.\ (\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 = \sigma_1 \cdot u_0 \wedge\\ &(\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n = \sigma_1 \cdot \sigma_w \rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) = \sigma_1 \cdot u_\mathsf{s}))\\ &\rightarrow \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot \mathbf{rec}(\boldsymbol{u_0}, \boldsymbol{u_\mathsf{s}}, \boldsymbol{x})).\end{aligned} \tag{IndAxSkoNew}$$

As a last step of preprocessing, we transform the quantifier-free segment of (IndAxSkoNew) into CNF, resulting in the two clauses

$$\begin{aligned}&\forall u_0 \in \mathbb{N} \forall u_\mathsf{s} \in \mathbb{N} \forall x \in \mathbb{N}.\\ &(\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n = \sigma_1 \cdot \sigma_w \vee\\ &\qquad \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot \mathsf{rec}(u_0, u_\mathsf{s}, x)) \wedge \qquad \text{(IndAxNew1)}\\ &\wedge (\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot 0 \neq \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) \neq \sigma_1 \cdot u_\mathsf{s} \vee\\ &\qquad \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot \mathsf{rec}(u_0, u_\mathsf{s}, x)). \qquad \text{(IndAxNew2)}\end{aligned}$$

Changing the input clauses (IndAxNew1) and (IndAxNew2) in this way enables us to not only track changes of variable x when applying induction, but also of the base case u_0 and step case u_s. The expression $\mathsf{rec}(u_0, u_\mathsf{s}, x)$ can then be computed as $f(x)$ where f is a function of the form

$$\begin{aligned}f(0) &= u_0\\ f(\mathsf{s}(n)) &= u_\mathsf{s}[f(n)].\end{aligned}$$

5.2 Recursive Synthesis in Theory

Based on the adaptions in Section 5.1, we nextly show how the recursive function for specification (SPD) can be synthesized.

The specification of the distributivity law is

$$\forall x_1, x_2, x_3 \in \mathbb{N}.\ \exists y \in \mathbb{N}.\ x_1 \cdot x_2 + x_1 \cdot x_3 = x_1 \cdot y. \tag{SPD}$$

Preprocessing works exactly like preprocessing for non-recursive synthesis. After negating (SPD), transformation into CNF, and adding the answer literal we have the following input clause

$$\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_3 \not\approx \sigma_1 \cdot y \vee \mathsf{ans}(y). \tag{SPD'}$$

Besides (IndAxNew1) and (IndAxNew2), the additionally used axioms are

$$\forall x \in \mathbb{N}. x + \mathsf{0} = x \tag{Add1}$$
$$\forall x \in \mathbb{N}. x \cdot \mathsf{0} = \mathsf{0} \tag{Mult1}$$
$$\forall x, n \in \mathbb{N}. x \cdot \mathsf{s}(n) = x \cdot n + x. \tag{Mult2}$$

Program derivation. We state the program derivation and nextly go through it step by step.

1. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_3 \not\approx \sigma_1 \cdot y \vee \mathsf{ans}(y)$ [input]
2. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{0} \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n \approx \sigma_1 \cdot \sigma_w \vee$
   $\vee\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x \approx \sigma_1 \cdot \mathsf{rec}(u_0, u_\mathsf{s}, x)$ [(IndAxNew1)]
3. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{0} \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) \not\approx \sigma_1 \cdot u_\mathsf{s} \vee$
   $\vee\ \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x \approx \sigma_1 \cdot \mathsf{rec}(u_0, u_\mathsf{s}, x)$ [(IndAxNew2)]
4. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{0} \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n \approx \sigma_1 \cdot \sigma_w \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_3))$ [BR 1, 2]
5. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{0} \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{s}(\sigma_n) \not\approx \sigma_1 \cdot u_\mathsf{s} \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_3))$ [BR 1, 3]
6. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{0} \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_n + \sigma_1 \not\approx \sigma_1 \cdot u_\mathsf{s} \vee$
   $\vee\ \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_3))$ [Sup 5, (Mult2)]
7. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{0} \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{0} \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_w + \sigma_1 \not\approx \sigma_1 \cdot u_\mathsf{s} \vee$
   $\vee\ \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_3))$ [Sup 4, 6]
8. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{0} \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \sigma_w + \sigma_1 \not\approx \sigma_1 \cdot u_\mathsf{s} \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_3))$ [F 7]
9. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{0} \not\approx \sigma_1 \cdot u_0 \vee \sigma_1 \cdot \mathsf{s}(\sigma_w) \not\approx \sigma_1 \cdot u_\mathsf{s} \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_3))$ [Sup 8, (Mult2)]
10. $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \mathsf{0} \not\approx \sigma_1 \cdot u_0 \vee \mathsf{ans}(\mathsf{rec}(u_0, \mathsf{s}(\sigma_w), \sigma_3))$ [ER 9]
11. $\sigma_1 \cdot \sigma_2 + \mathsf{0} \not\approx \sigma_1 \cdot u_0 \vee \mathsf{ans}(\mathsf{rec}(u_0, \mathsf{s}(\sigma_w), \sigma_3))$ [Sup 10, (Mult1)]
12. $\sigma_1 \cdot \sigma_2 \not\approx \sigma_1 \cdot u_0 \vee \mathsf{ans}(\mathsf{rec}(u_0, \mathsf{s}(\sigma_w), \sigma_3))$ [Sup 11, (Add1)]

13. $\mathsf{ans}(\mathsf{rec}(\sigma_2, \mathsf{s}(\sigma_w), \sigma_3))$ [ER 12]

14. $\square$ [answer literal removal 13]

Going through program derivation. This derivation is analogous to the one in Chapter 3, with additional tracking of changes of the variables by the answer literal.

Together with the input clause 1, we also add the two clauses of our modified induction axioms, (IndAxNew1) and (IndAxNew2), to the clause set. We then resolve the input clause 1 with clause 2, resulting in clause 4, and a second time with clause 3 resulting in clause 5. This is the step where the rec-function firstly appears inside the answer literal which happens due to the characteristics of the binary resolution rule.

In order to unify the literals $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot \sigma_3 = \sigma_1 \cdot y$ and $\sigma_1 \cdot \sigma_2 + \sigma_1 \cdot x = \sigma_1 \cdot \mathsf{rec}(u_0, u_\mathsf{s}, x)$, the substitution $\theta_1 : \{x \mapsto \sigma_3, y \mapsto \mathsf{rec}(u_0, u_\mathsf{s}, \sigma_3)\}$ is used. Due to the fact that the rec function is considered to be computable, this results in a substitution of the answer literal to $\mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_3))$, as can be seen in clauses 4 and 5.

Nextly, the same steps occur as in the derivation of Chapter 3. The answer literal changes again when applying equality resolution on clause 9. Here, the substitution $\theta_2 : \{u_\mathsf{s} \mapsto \mathsf{s}(\sigma_w)\}$ is used, which results in the answer literal $\mathsf{ans}(\mathsf{rec}(u_0, \mathsf{s}(\sigma_w), \sigma_3))$. In clause 12, equality resolution is applied again, resulting in the answer literal changing to $\mathsf{ans}(\mathsf{rec}(\sigma_2, \mathsf{s}(\sigma_w), \sigma_3))$. Additionally, the clause containing only the answer literal is derived. As a last step, the answer literal is removed resulting in the empty clause and the derivation terminates. The skolem constants are then mapped to their stored variables, resulting in the answer literal

$$\mathsf{ans}(\mathsf{rec}(x_2, \mathsf{s}(w), x_3)). \tag{5.2}$$

Constructing recursive function from given answer literal. The next task is to construct the recursive function from (5.2). Here we have to remember what each of the arguments in (5.2) stands for. The third argument in the rec-function stands for the input variable. Therefore the recursive function, denoted with f, has the input variable x_3. Nextly, the first argument was originally the variable found for the base case where the input is $\mathsf{0}$. Therefore, we know that the function f has the output x_2 for the input $\mathsf{0}$. As a last step, for given n we know that the function returns the value w, one could thus write $f(n)$. Therefore, we know that $f(\mathsf{s}(n)) = \mathsf{s}(w) = \mathsf{s}(f(n))$. Putting this all together we conclude with the function $f(x_3)$ for (5.2), where

$$\begin{aligned} f(\mathsf{0}) &= x_2, \\ f(\mathsf{s}(n)) &= \mathsf{s}(f(n)), \end{aligned}$$

which can be easily validated to be $x_2 + x_3$.

5.3 Recursive Synthesis in Practice using Vampire

Laura Kovács from TU Wien and Andrei Voronkov from the University of Manchester, and their respective research groups, develop the first-order theorem prover Vampire for reasoning with first-order theories with equality, using the superpostion calculus, [KV13].

The process described in Section 5.1 was implemented in VAMPIRE by Petra Hozzová and Daneshvar Amrollahi. In chapter 7 more examples will be presented not only over term algebras of natural numbers but also natural lists and natural binary trees. Several of these examples work in practice, which can be looked into in Table 7.1. These examples and derivations have been my contributions.

For the proving example of the law of distributivity, Petra Hozzová successfully found the following proving configuration in VAMPIRE.

```
-forced_options av=off:bd=preordered:bce=on:flr=on:fsr=off:lma=on:nwc=2.0
:sp=occurrence:urr=ec_only:ind=struct:indu=off:qa=synthesis:tgt=off:erd=off
:updr=off:indc=goal_plus:indgenss=2:indgen=on:indmd=2:drc=off:to=lpo_1010
```

The command calls upon an input file that contains the specification (SPD), as well as the axioms defining the function symbols " + " and " · ", written in SMT-Lib syntax [Cok13]:

```
    (set-logic UFDT)

; TYPE OF NATURAL NUMBERS
(declare-datatypes ((nat 0)) (((zero) (s (s0 nat)))))

(declare-fun add (nat nat) nat)

(declare-fun mult (nat nat) nat)

;; NATURAL NUMBER AXIOMS
; add base
(assert (forall ((x nat)) (= x (add x zero))))
; add step
(assert (forall ((x nat) (n nat)) (= (add x (s n)) (s (add x n
   )))))
; mult base
(assert (forall ((x nat)) (= zero (mult x zero))))
; mult step
(assert (forall ((x nat) (n nat)) (= (mult x (s n)) (add (mult
    x n) x))))


; SPECIFICATION
(assert-not(
        forall ((x nat) (y nat) (z nat))
                (exists ((w nat))
                        (=
                                (add (mult x y) (mult x z))
```

```
                                                (mult x w)
                                    )
                            )
                    )
)

(check-sat)
```

In the following we will state the full output of VAMPIRE. The idea here is not to fully trace back everything VAMPIRE is doing but to (i) highlight the important steps that were also discussed in the theoretic proof of Section 5.2 and (ii) to show the full scale of how a theorem prover works. The text highlighted in red corresponds to the proof steps of Section 5.2.

```
% Running in auto input_syntax mode. Trying SMTLIB2
% Refutation found. Thanks to Tanya!
% SZS status Unsatisfiable for distributivity
% Inputs for synthesis:
6. ~! [X1 : 'nat()',X0 : 'nat()',X2 : 'nat()'] : ? [X3 : 'nat()'] :
   mult(X0,X3) = add(mult(X0,X1),mult(X0,X2)) [negated conjecture 5]
% Recursive function definitions:
rf3523(zero) = X1
rf3523(s(X7)) = s(rf3523(X7))
% SZS answers Tuple [[rf3523(X2)]|_] for distributivity
% SZS output start Proof for distributivity
1. ! [X0 : 'nat()'] : add(X0,zero) = X0 [input]
2. ! [X1 : 'nat()',X0 : 'nat()'] : s(add(X0,X1)) = add(X0,s(X1)) [
   input]
3. ! [X0 : 'nat()'] : zero = mult(X0,zero) [input]
4. ! [X1 : 'nat()',X0 : 'nat()'] : add(mult(X0,X1),X0) = mult(X0,s(
   X1)) [input]
5. ! [X1 : 'nat()',X0 : 'nat()',X2 : 'nat()'] : ? [X3 : 'nat()'] :
   mult(X0,X3) = add(mult(X0,X1),mult(X0,X2)) [input]
6. ~! [X1 : 'nat()',X0 : 'nat()',X2 : 'nat()'] : ? [X3 : 'nat()'] :
   mult(X0,X3) = add(mult(X0,X1),mult(X0,X2)) [negated conjecture 5]
10. ! [X3 : 'nat()'] : ~(mult(sK2_in,X3) =
   add(mult(sK2_in,sK1_in),mult(sK2_in,sK3_in)) & ans0(X3)) [answer literal
   with input var skolemisation 6]
11. ! [X0 : 'nat()'] : ~(add(mult(sK2_in,sK1_in),mult(sK2_in,sK3_in)
   ) = mult(sK2_in,X0) & ans0(X0)) [rectify 10]
12. ! [X0 : 'nat()',X1 : 'nat()'] : add(mult(X1,X0),X1) = mult(X1,s(
   X0)) [rectify 4]
13. ! [X0 : 'nat()',X1 : 'nat()'] : s(add(X1,X0)) = add(X1,s(X0)) [
   rectify 2]
14.   ! [X0 : 'nat()'] : (add(mult(sK2_in,sK1_in),mult(sK2_in,sK3_in
   )) != mult(sK2_in,X0) | ~ans0(X0)) [ennf transformation 11]
15. add(mult(sK2_in,sK1_in),mult(sK2_in,sK3_in)) != mult(sK2_in,X0)
```

```
   | ~ans0(X0) [cnf transformation 14]
16. add(mult(X1,X0),X1) = mult(X1,s(X0)) [cnf transformation 12]
17. zero = mult(X0,zero) [cnf transformation 3]
18. s(add(X1,X0)) = add(X1,s(X0)) [cnf transformation 13]
19. add(X0,zero) = X0 [cnf transformation 1]
34. ? [X7 : 'nat()'] : ? [X8 : 'nat()'] : ! [X9 : 'nat()',X5 : 'nat
    ()'] : ! [X10 : 'nat()'] : ((add(mult(sK2_in,sK1_in),mult(sK2_in,
    zero)) = mult(sK2_in,X5) & (add(mult(sK2_in,sK1_in),mult(sK2_in,
    X7)) = mult(sK2_in,X8) => add(mult(sK2_in,sK1_in),mult(sK2_in,s(
    X7))) = mult(sK2_in,X9))) => add(mult(sK2_in,sK1_in),mult(sK2_in,
    X10)) = mult(sK2_in,rec10(X5,X9,X10))) [structural induction
    hypothesis]
35. ? [X7 : 'nat()'] : ? [X8 : 'nat()'] : ! [X9 : 'nat()',X5 : 'nat
    ()'] : ! [X10 : 'nat()'] : (add(mult(sK2_in,sK1_in),mult(sK2_in,
    X10)) = mult(sK2_in,rec10(X5,X9,X10)) | (add(mult(sK2_in,sK1_in),
    mult(sK2_in,zero)) != mult(sK2_in,X5) | (add(mult(sK2_in,sK1_in),
    mult(sK2_in,s(X7))) != mult(sK2_in,X9) & add(mult(sK2_in,sK1_in),
    mult(sK2_in,X7)) = mult(sK2_in,X8)))) [ennf transformation 34]
36. add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)) = mult(sK2_in,sK12) |
     add(mult(sK2_in,sK1_in),mult(sK2_in,zero)) != mult(sK2_in,X5) |
    add(mult(sK2_in,sK1_in),mult(sK2_in,X10)) = mult(sK2_in,rec10(X5,
    X9,X10)) [cnf transformation 35]
37. mult(sK2_in,X9) != add(mult(sK2_in,sK1_in),mult(sK2_in,s(sK11)))
     | add(mult(sK2_in,sK1_in),mult(sK2_in,zero)) != mult(sK2_in,X5)
    | add(mult(sK2_in,sK1_in),mult(sK2_in,X10)) = mult(sK2_in,rec10(
    X5,X9,X10)) [cnf transformation 35]
38. add(mult(sK2_in,sK1_in),mult(sK2_in,s(sK11))) != mult(sK2_in,X1) |
    mult(sK2_in,X0) != add(mult(sK2_in,sK1_in),mult(sK2_in,zero)) |
     ans0(rec10(X0,X1,sK3_in)) [resolution 37,15]
39. add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)) = mult(sK2_in,sK12) |
     mult(sK2_in,X0) != add(mult(sK2_in,sK1_in),mult(sK2_in,zero)) |
    ~ans0(rec10(X0,X1,sK3_in)) [resolution 36,15]
107. mult(sK2_in,X0) != add(mult(sK2_in,sK1_in),zero) | add(mult(
    sK2_in,sK1_in),mult(sK2_in,sK11)) = mult(sK2_in,sK12) | ~ans0(
    rec10(X0,X1,sK3_in)) [forward demodulation 39,17]
108. mult(sK2_in,sK1_in) != mult(sK2_in,X0) | add(mult(sK2_in,sK1_in
    ),mult(sK2_in,sK11)) = mult(sK2_in,sK12) | ~ans0(rec10(X0,X1,
    sK3_in)) [forward demodulation 107,19]
109. mult(sK2_in,X1) != add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11
    ),sK2_in)) | mult(sK2_in,X0) != add(mult(sK2_in,sK1_in),mult(
    sK2_in,zero)) | ~ans0(rec10(X0,X1,sK3_in)) [forward demodulation
    38,16]
110. mult(sK2_in,X0) != add(mult(sK2_in,sK1_in),zero) | mult(sK2_in,
    X1) != add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),sK2_in)) | ~
    ans0(rec10(X0,X1,sK3_in)) [forward demodulation 109,17]
111. mult(sK2_in,X1) != add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11
    ),sK2_in)) | mult(sK2_in,sK1_in) != mult(sK2_in,X0) | ~ans0(rec10
    (X0,X1,sK3_in)) [forward demodulation 110,19]
736. add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)) = mult(sK2_in,sK12)
    | ~ans0(rec10(sK1_in,X0,sK3_in)) [equality resolution 108]
```

```
1456. add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),sK2_in)) != add(
   mult(sK2_in,X0),sK2_in) | mult(sK2_in,sK1_in) != mult(sK2_in,X1)
   | ~ans0(rec10(X1,s(X0),sK3_in)) [superposition 111,16]
18325. add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),sK2_in)) != add
   (add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),sK2_in) | mult(sK2_in
   ,sK1_in) != mult(sK2_in,sK1_in) | ~ans0(rec10(sK1_in,s(sK12),sK3_in)) [
   superposition 1456,736]
18471. add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),sK2_in)) != add
   (add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),sK2_in) | ~ans0(rec10
   (sK1_in,s(sK12),sK3_in)) [trivial inequality removal 18325]
18771. ! [X0 : 'nat()'] : (add(mult(sK2_in,sK1_in),add(mult(sK2_in,
   sK11),zero)) = add(add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),
   zero) & (add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),X0)) = add
   (add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),X0) => add(mult(
   sK2_in,sK1_in),add(mult(sK2_in,sK11),s(X0))) = add(add(mult(
   sK2_in,sK1_in),mult(sK2_in,sK11)),s(X0)))) => ! [X1 : 'nat()'] :
   add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),X1)) = add(add(mult
   (sK2_in,sK1_in),mult(sK2_in,sK11)),X1) [structural induction
   hypothesis]
18772. ! [X1 : 'nat()'] : add(mult(sK2_in,sK1_in),add(mult(sK2_in,
   sK11),X1)) = add(add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),X1) |
    ? [X0 : 'nat()'] : (add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11
   ),zero)) != add(add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),zero)
   | (add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),s(X0))) != add(
   add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),s(X0)) & add(mult(
   sK2_in,sK1_in),add(mult(sK2_in,sK11),X0)) = add(add(mult(sK2_in,
   sK1_in),mult(sK2_in,sK11)),X0))) [ennf transformation 18771]
18773. add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),sK3519)) = add(
   add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),sK3519) | add(mult(
   sK2_in,sK1_in),add(mult(sK2_in,sK11),zero)) != add(add(mult(
   sK2_in,sK1_in),mult(sK2_in,sK11)),zero) | add(mult(sK2_in,sK1_in)
   ,add(mult(sK2_in,sK11),X1)) = add(add(mult(sK2_in,sK1_in),mult(
   sK2_in,sK11)),X1) [cnf transformation 18772]
18774. add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),s(sK3519))) !=
   add(add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),s(sK3519)) | add(
   mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),zero)) != add(add(mult(
   sK2_in,sK1_in),mult(sK2_in,sK11)),zero) | add(mult(sK2_in,sK1_in)
   ,add(mult(sK2_in,sK11),X1)) = add(add(mult(sK2_in,sK1_in),mult(
   sK2_in,sK11)),X1) [cnf transformation 18772]
18775. add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),s(sK3519))) !=
   add(add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),s(sK3519)) | add(
   mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),zero)) != add(add(mult(
   sK2_in,sK1_in),mult(sK2_in,sK11)),zero) | ~ans0(rec10(sK1_in,s(
   sK12),sK3_in)) [generalized induction hyperresolution
   18471,18774]
18776. add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),sK3519)) = add(
   add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),sK3519) | add(mult(
   sK2_in,sK1_in),add(mult(sK2_in,sK11),zero)) != add(add(mult(
   sK2_in,sK1_in),mult(sK2_in,sK11)),zero) | ~ans0(rec10(sK1_in,s(
   sK12),sK3_in)) [generalized induction hyperresolution
```

```
   18471,18773]
18811. add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)) != add(mult(sK2_in
   ,sK1_in),add(mult(sK2_in,sK11),zero)) | add(mult(sK2_in,sK1_in),
   add(mult(sK2_in,sK11),sK3519)) = add(add(mult(sK2_in,sK1_in),mult
   (sK2_in,sK11)),sK3519) | ~ans0(rec10(sK1_in,s(sK12),sK3_in)) [
   forward demodulation 18776,19]
18812. add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)) != add(mult(sK2_in
   ,sK1_in),mult(sK2_in,sK11)) | add(mult(sK2_in,sK1_in),add(mult(
   sK2_in,sK11),sK3519)) = add(add(mult(sK2_in,sK1_in),mult(sK2_in,
   sK11)),sK3519) | ~ans0(rec10(sK1_in,s(sK12),sK3_in)) [forward
   demodulation 18811,19]
18813. add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),sK3519)) = add(
   add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),sK3519) | ~ans0(rec10(
   sK1_in,s(sK12),sK3_in)) [trivial inequality removal 18812]
18814. add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),s(sK3519))) !=
   s(add(add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),sK3519)) | add(
   mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),zero)) != add(add(mult(
   sK2_in,sK1_in),mult(sK2_in,sK11)),zero) | ~ans0(rec10(sK1_in,s(
   sK12),sK3_in)) [forward demodulation 18775,18]
18815. s(add(add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),sK3519)) !=
   add(mult(sK2_in,sK1_in),s(add(mult(sK2_in,sK11),sK3519))) | add(
   mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),zero)) != add(add(mult(
   sK2_in,sK1_in),mult(sK2_in,sK11)),zero) | ~ans0(rec10(sK1_in,s(
   sK12),sK3_in)) [forward demodulation 18814,18]
18816. s(add(add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),sK3519)) !=
   s(add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),sK3519))) | add(
   mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),zero)) != add(add(mult(
   sK2_in,sK1_in),mult(sK2_in,sK11)),zero) | ~ans0(rec10(sK1_in,s(
   sK12),sK3_in)) [forward demodulation 18815,18]
18817. add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)) != add(mult(sK2_in
   ,sK1_in),add(mult(sK2_in,sK11),zero)) | s(add(add(mult(sK2_in,
   sK1_in),mult(sK2_in,sK11)),sK3519)) != s(add(mult(sK2_in,sK1_in),
   add(mult(sK2_in,sK11),sK3519))) | ~ans0(rec10(sK1_in,s(sK12),
   sK3_in)) [forward demodulation 18816,19]
18818. add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)) != add(mult(sK2_in
   ,sK1_in),mult(sK2_in,sK11)) | s(add(add(mult(sK2_in,sK1_in),mult(
   sK2_in,sK11)),sK3519)) != s(add(mult(sK2_in,sK1_in),add(mult(
   sK2_in,sK11),sK3519))) | ~ans0(rec10(sK1_in,s(sK12),sK3_in)) [
   forward demodulation 18817,19]
18819. s(add(add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),sK3519)) !=
   s(add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),sK3519))) | ~ans0
   (rec10(sK1_in,s(sK12),sK3_in)) [trivial inequality removal 18818]
18820. add(mult(sK2_in,sK1_in),add(mult(sK2_in,sK11),sK3519)) != add
   (add(mult(sK2_in,sK1_in),mult(sK2_in,sK11)),sK3519) | ~ans0(rec10
   (sK1_in,s(sK12),sK3_in)) [term algebras injectivity 18819]
18877. ~ans0(rec10(sK1_in,s(sK12),sK3_in)) [unit resulting resolution
   18813,18820]
18878. ans0(X0) [answer literal]
18879. $false [unit resulting resolution 18878,18877]
% SZS output end Proof for distributivity
```

```
% ------------------------------
% Version: Vampire 4.8 (commit cc590a820 on 2024-02-04 18:01:23
    +0100)
% Linked with Z3 4.12.2.0 e417f7d78509b2d0c9ebc911fee7632e6ef546b6
    z3-4.8.4-7517-ge417f7d78
% Termination reason: Refutation

% Memory used [KB]: 15971
% Time elapsed: 0.852 s
% ------------------------------
% ------------------------------
```

Going through main steps. We now discuss the output highlighted in red.

1. In line 6 the negated specification is highlighted. The exclamation mark "!" denotes a universal quantifier and the question mark "?" denotes an existential quantifier. The tilde symbol "$\sim$" denotes a negation, $\neg$. Further, we see the input variables $X0, X1$ and $X2$ and the output variable $X3$ with their respective types.

2. In line 10 the specification is fully preprocessed, including the answer literal, containing the output variable $X3$.

3. In line 38 we see the resolution of the induction axiom together with the input clause which results in the first appearance of an answer literal with a rec-term.

4. In line 736 the rec-term changes, the skolem constant $sK1\_in$ is substituted for the variable $X0$.

5. In line 18325 the rec-term changes again, here the term $\mathsf{s}(sK\_12)$ is substituted for the variable $X0$.

6. The clause containing only the answer literal is derived in line 18877, which evokes termination of the derivation.

In conclusion, we see that the automatic proof of the Vampire theorem prover follows a similar guideline as the manually deducted theoretic proof of Section 5.2.

6 Synthesis of Recursive Programs in Saturation

In this chapter, our goal is to formalize the program synthesis process that was described based on the distributivity example in Chapter 5. In Section 6.1 we define the concepts of inductive structures, inductive proofs, and recursive functions, [GJ95]. In Section 6.2 we explain how one can construct recursive functions given an inductive proof over an inductive structure [HAH+24]. Section 6.1 and Section 6.2 are mostly based on the inductive structure of natural numbers. In Section 6.3 we will explain how synthesis of recursive functions works for the inductive structures of lists and binary trees.

6.1 Inductive Structures

The idea of being able to synthesize recursive functions from first-order logic specifications relies on the one-to-one correspondence between induction and recursion. For this we will introduce the following concept.

Definition 6.1.1 (Inductiv Structure). A set $\mathcal{S}$ is called an *inductive structure*, notated as $\mathcal{S} = \mathsf{ind}(\mathcal{C}, \mathcal{O})$, where $\mathcal{C}$ denotes a set of constants and $\mathcal{O}$ a set of functions, together with their arity if the following properties hold.

1. Every constant $c \in \mathcal{C}$ is in $\mathcal{S}$.
2. For every operator $f \in \mathcal{O}$ with arity n and $a_1, ..., a_n \in \mathcal{S}$ it holds that $f(a_1, ..., a_n) \in \mathcal{S}$.
3. For every element $s \in \mathcal{S}$ it holds that either $s \in \mathcal{C}$ or there are elements $a_1, ..., a_n \in \mathcal{S}$, a function $f \in \mathcal{O}$ with arity n such that $s = f(a_1, ..., a_n)$.

The elements in $\mathcal{C}$ and $\mathcal{O}$ are also called *constructors*.

Remark 6.1.2 The name *inductive structure* comes from the fact that these structures are defined *inductively* using constants and operators.

Example 6.1.3 We have already encountered an example of an inductive structure, namely the theory of natural numbers denoted as $\mathbb{N}$. Additionally to the predicate symbol of equality " = ", defined by the axioms given in Section 2.3, it consists of the function symbols $\mathsf{0}$ and s for which the following two axioms hold

$$\forall x.\ \mathsf{s}(x) \neq \mathsf{0} \tag{Nat1}$$

$$\forall x, y.\ \mathsf{s}(x) = \mathsf{s}(y) \rightarrow x = y. \tag{Nat2}$$

Therefore, the inductive structure of the natural numbers is induced by the constant $\mathsf{0}$ and the operator s with arity 1 and can be written as $\mathbb{N} = \mathsf{ind}(\{\mathsf{0}\}, \{\mathsf{s}\})$. Following Definition 6.1.1 we can construct the elements of $\mathbb{N}$ in the following way.

1. $\mathsf{0} \in \mathcal{C} \rightarrow \mathsf{0} \in \mathbb{N}$
2. $\mathsf{0} \in \mathbb{N}, \mathsf{s} \in \mathcal{O} \rightarrow \mathsf{s}(\mathsf{0}) \in \mathbb{N}$
3. $\mathsf{s}(\mathsf{0}) \in \mathbb{N}, \mathsf{s} \in \mathcal{O} \rightarrow \mathsf{s}(\mathsf{s}(\mathsf{0})) \in \mathbb{N}$
4. ...

Further, it follows from item 3 of Definition 6.1.1, that only elements constructed by $\mathsf{0}$ and s are contained in $\mathbb{N}$. Therefore, we can express the set of natural numbers as

$$\mathbb{N} = \{\mathsf{0}, \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\mathsf{0})), \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))), ...\}.$$

The idea of having inductive structures is that one can easily establish whether some property holds, by *proving with induction*.

Definition 6.1.4 (Proof by Induction). Let $\mathcal{S} = \mathsf{ind}(\mathcal{C}, \mathcal{O})$ be an inductive structure and $P : \mathcal{S} \rightarrow \{\top, \bot\}$ a property. If

1. for each constant $c \in \mathcal{C}$, $P(c)$ is true and
2. for each function $f \in \mathcal{O}$ with arity n and elements $s_1, ..., s_n$ for which property P is true it follows that $P(f(s_1, ..., s_n))$ is true as well,

then the following formula is *valid*

$$\forall x \in \mathcal{S}.P(x).$$

Definition 6.1.5 (Proof by Induction over $\mathbb{N}$). For the inductive structure of the natural numbers $\mathbb{N}$, proof by induction simplifies to the following. Let $P : \mathbb{N} \rightarrow \{\top, \bot\}$ be an arbitrary property. If

1. $P(\mathsf{0})$ holds
2. and for each element $x \in \mathbb{N}$ it holds that $P(x) \rightarrow P(\mathsf{s}(x))$

then

$$\forall x \in \mathbb{N}.P(x)$$

is valid.

As a next step, we define the property of unique readability over inductive structures. This property ensures that each element of an inductive structure can be expressed in *exactly* one way. We will then use this property for expressing *unique* recursive functions over inductive structures.

Definition 6.1.6 (Unique Readability). An inductive structure $\mathcal{S} = \mathsf{ind}(\mathcal{C}, \mathcal{O})$ has the property of *unique readability* if for every $b \in \mathcal{S}$ exactly one of the following two properties P_1, P_2 hold:

1. $P_1(b) : b \in \mathcal{C}$ or
2. $P_2(b)$: There are *unique* elements $s_1, ..., s_n \in \mathcal{S}$ and a *unique* operator $f \in \mathcal{O}$ with arity n such that $b = f(s_1, ..., s_n)$ and no element in $\mathcal{C}$, other than b, is of the form $f(s_1, ..., s_n)$.

We prove that the property of unique readability holds for the natural numbers.

Lemma 6.1.7 *The inductive structure of the natural numbers* $\mathbb{N} = \mathsf{ind}(\{\mathsf{0}\}, \{\mathsf{s}\})$ *has the property of unique readability.*

Proof. We prove by induction. Following Definition 6.1.6 we want to prove that the following property holds for all elements in $\mathbb{N}$:

$$P(x) : P_1(x) \vee P_2(x).$$

The *base case* is easily established: Because $\mathsf{0} \in \mathcal{C}$ also $P(\mathsf{0})$ holds.
For the *step case*, we assume that $P(x)$ for arbitrary $x \in \mathbb{N}$ holds. There are two cases to distinguish.
Case 1. Assume that $P_1(x)$ is true. This means that $x \in \mathcal{C} = \{\mathsf{0}\}$ from which $x = \mathsf{0}$ follows. Using the substitution axiom for s, (2.7), it follows that $\mathsf{s}(x) = \mathsf{s}(\mathsf{0})$. The set of operators of the natural numbers consists only of s with arity 1. Therefore, there is an element in $\mathbb{N}$, $\mathsf{0}$, and a unique operator, s, such that $\mathsf{s}(x) = \mathsf{s}(\mathsf{0})$. Assume there is an element x' in $\mathbb{N}$ different from $\mathsf{0}$ such that $\mathsf{s}(x) = \mathsf{s}(x')$ holds. Using the injectivity axiom (Nat2) it follows that $x = x'$ and therefore it follows from the transitivity axiom of the equality sign that also $x' = \mathsf{0}$. This shows that $\mathsf{0}$ is the unique element such that $\mathsf{s}(x) = \mathsf{s}(\mathsf{0})$.
From the definition of an inductive structure it follows that for any element $b' \in \mathbb{N}$ it either holds that

1. $b' = \mathsf{0}$ or
2. there is $n \in \mathbb{N}$ such that $b' = \mathsf{s}(n)$.

Axiom (Nat1) states that $\mathsf{0} \neq \mathsf{s}(x)$ and Axiom (Nat2) states that if $\mathsf{s}(n) = \mathsf{s}(x)$ it must follow that $n = x$. This proves that no element other than $\mathsf{s}(x)$ can be written as $\mathsf{s}(\mathsf{0})$.

We have proven that $P_2(\mathsf{s}(x))$ holds and therefore also $P(\mathsf{s}(x))$ holds.
Case 2. If Case 1 does not hold, $\neg P_1(x)$, it immediately follows that $P_2(x)$ must be true. There is only one operator in $\mathcal{O}$: s of arity 1. Therefore it follows that there is a unique element, s_x, and a unique operator, s, such that $x = \mathsf{s}(s_x)$. Using again the substitution axiom (2.7) we get $\mathsf{s}(x) = \mathsf{s}(\mathsf{s}(s_x))$. In particular, there is an element, $\mathsf{s}(s_x)$ and a unique (the only) operator, s such that $\mathsf{s}(x) = \mathsf{s}(\mathsf{s}(s_x))$. Assume there is another s'_x in $\mathbb{N}$ such that $x = \mathsf{s}(s'_x)$. This means that $\mathsf{s}(s_x) = \mathsf{s}(s'_x)$. Using Axiom (Nat2) it follows that $s_x = s'_x$. This shows the uniqueness of element s_x.

The fact that there is no other element in $\mathbb{N}$ that can be written as $\mathsf{s}(s_x)$ follows from the same argument given in Case 1.

This shows that $P_2(\mathsf{s}(x))$ holds.

We have shown the precondition for induction in Definition 6.1.5, therefore we can deduce

$$\forall x \in \mathbb{N}.P(x),$$

which completes the proof. □

As a next step, we state the following lemma, which defines recursive functions over natural numbers.

Lemma 6.1.8 (Recursive functions over $\mathbb{N}$). *For an arbitrary set A let $G_\mathsf{s} : A \to A$ be a function and let a_0 be an element of A. Then there exists a* unique *function $f : \mathbb{N} \to A$ such that the following properties hold*

$$f(\mathsf{0}) = a_0$$
$$\forall k \in \mathbb{N}.f(\mathsf{s}(k)) = G_\mathsf{s}(f(k)).$$

Proof. See Proof of Lemma 1.1.30 in [GJ95]. □

Remark 6.1.9 We call functions constructed as in Lemma 6.1.8 *recursive functions over the natural numbers*. Note, that the recursive function f is uniquely defined by the pair (a_0, G_s) in an *inductive* way.

Remark 6.1.10 The result of Lemma 6.1.8 can be generalized to arbitrary inductive structures with unique readability, see Lemma 1.1.31 in [GJ95].

Remark 6.1.11 Going back to Chapter 5 we have seen that after the termination of the proof derivation, the answer literal contained the term

$$\mathsf{rec}(x_2, \mathsf{s}(w), x_3).$$

This corresponds uniquely to the recursive function f defined by the tuple $(x_2, \mathsf{s}(.))$, where the value for input x_3 is computed.

We have established that proof by induction is very useful to showcase certain properties of inductive structures. Further, we can uniquely define recursive function over inductive structures with unique readabilty. When proving by induction over formulas with a special structure, the pair (a_0, G_s) can be used to define a recursive function.

As a next step we formalize the process of translating inductive proofs to recursive functions over natural numbers.

6.2 Constructing Recursive Functions from Inductive Proofs

We explain the following process for the inductive structure on natural numbers to get a better intuition. In Section 6.3 we explain how this works for the inductive structures of lists and binary trees.

We start with a specification of the form

$$\forall \overline{x}\ \exists y.F[\overline{x}, y]. \tag{6.1}$$

The idea is the following: when a clause containing a literal of the form $\neg G[\sigma_x, y]$ is derived, where the variable y is the output variable of (6.1), the induction axiom can be applied using the formula

$$\forall x\ \exists y.G[x, y]. \tag{6.2}$$

The induction axiom with the plugged-in formula, (6.2), is next preprocessed. We have described this in more detail in Section 5.1, for a specific formula. Preprocessing entailed transformation into PNF with the right order of the quantified variable and introduced the skolem function that we denoted with rec. This results in a formula of the form

$$\begin{aligned}\mathsf{cnf}((G[0, u_0] \ \wedge (G[\sigma_n, \sigma_v] \rightarrow G[\mathsf{s}(\sigma_n), u_\mathsf{s}])) \\ \rightarrow G[x, \mathsf{rec}(u_0, u_\mathsf{s}, x)]).\end{aligned} \tag{MagAxNat}$$

Remark 6.2.1 We call formulas of the form (MagAxNat) *magic axioms*. This is done in order to shorten notation and highlight the main mechanism of synthesizing recursive functions.

We use the literal $G[x, \mathsf{rec}(u_0, u_\mathsf{s}, x)]$ contained in all clauses of (MagAxNat) to resolve the literal $\neg G[\sigma_x, y]$ with the substitution $\theta : \{x \mapsto \sigma_x, y \mapsto \mathsf{rec}(u_0, u_\mathsf{s}, \sigma_x)\}$. This application of θ then also results in substituting the output, variable y, inside the answer literal to $\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_x)$. Resolving further literals in (MagAxNat) results in substitutions of variables u_0 and u_s. If at some stage the proof terminates, the content of the answer literal is postprocessed into a program consisting of recursive functions.

Remark 6.2.2 Several applications of induction during the proof can result in several rec-terms nested inside of each other. After successful termination, the content of the answer literal is unrolled inside out resulting in nested recursive functions. This will be demonstrated with concrete examples (Examples 7.1.1 and 7.2.3) in Chapter 7.

Remark 6.2.3 If the formula (6.2), where induction is applied upon, consists of a unit clause (which was the case with the distributivity example in Section (5.2)), the transformation of the formula (MagAxNat) into CNF results in two clauses. If G is a clause

containing more literals or even consists of more clauses, the formula (MagAxNat) turns into a vastly bigger clause set, which will be illustrated with concrete examples in Chapter 7. Examples 7.2.4 and 7.3.1 use induction on a formula of the form $\{\{l_1, l_2\}, \{l_1', l_2'\}\}$. The added clause set then contains a total of 12 clauses, which already highly increases the size of the derivation.

Adapted Inference Rules of the Superposition Calculus. Additionally to appearances of rec-terms that are translated into recursive functions, we introduce two new rules of the superposition calculus in Figure 6.1. These two rules allow for if − then − else−constructors to appear inside of rec-terms. The rules can be applied if the rec-terms in the two clauses have the same input arguments r_0 and r and have two unifiable expressions r_s and r_s' as their second argument. The if − then − else−constructor then appears in the second argument of the rec-term. If the derivation is indeed successful, the program then contains recursive functions with if − then − else− conditions in the step case.

Remark 6.2.4 The derivations of the Examples 7.1.2, 7.1.3, 7.2.4 and 7.3.1 in Chapter 7 make use of the adapted rules given in Figure 6.1.

Binary resolution with condition inside rec-term (BR"):

$$\frac{\underline{L} \vee C \vee \mathsf{ans}(\mathsf{rec}(r_0, r_s, r)) \quad \underline{\neg L'} \vee C' \vee \mathsf{ans}(\mathsf{rec}(r_0, r_s', r))}{(D \vee C \vee C' \vee \mathsf{ans}(\mathsf{rec}(r_0, \mathsf{if}\ L\ \mathsf{then}\ r_s'\ \mathsf{else}\ r_s, r)))\theta}$$

where (θ, D) is a computable unifier of L, L' wrt. $\mathsf{rec}(r_0, \mathsf{if}\ L\ \mathsf{then}\ r_s'\ \mathsf{else}\ r_s, r))$

Superposition with condition inside rec-term (Sup"):

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

where (θ, D) is a computable unifier of s, s' wrt. $\mathsf{rec}(r_0, \mathsf{if}\ s = t\ \mathsf{then}\ r_s'\ \mathsf{else}\ r_s, r))$.

Figure 6.1: Additional rules of the superposition calculus Sup for recursive synthesis. The expressions s, s', t, t' denote terms, L denotes a literal, C and D denote clauses.

The next theorem ensures correctness of the introduced framework for recursive synthesis.

Theorem 6.2.5 (Correctness of Superposition with Recursive Synthesis). *Assume that for a given specification*

$$\forall \overline{x}\ \exists y.F[\overline{x}, y] \tag{6.3}$$

the clause containing only the answer literal $\mathsf{ans}(r[\overline{\sigma}])$ *has been derived from the set of clauses*

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

where $M_1, ..., M_n$ *denote different instances of magic axioms and* $r[\sigma]$ *is computable. We write* $r^R[\overline{x}]$ *for the term where for each* rec*-term coming from a different magic axiom the unique recursive function has been substituted. The expression* $r^R[\overline{x}]$ *is then a witness for* y *for* (6.3).

Proof. A detailed proof can be found in [HAH+24], page 23. □

6.3 Magic Axioms for Different Inductive Structures

Besides natural numbers, we will also derive examples over different inductive structures, i.e. natural lists and natural binary trees. In the following we will shortly explain the magic axioms for these inductive structures. We refer to Section 8 in [HAH+24] for the general version of the structural induction axiom and its respective translation into a recursive function.

6.3.1 Natural Lists

The inductive structure of natural lists consists of the constructors nil of arity 0 and $\mathsf{cons} : \mathbb{N} \times \mathbb{L} \to \mathbb{L}$ of arity 2. Therefore the *structural induction axiom over lists* looks like

$$\begin{aligned}(F[\mathsf{nil}] \wedge \forall n \in \mathbb{N}, l \in \mathbb{L}.(F[l] \to F[\mathsf{cons}(n, l)])) \\ \to \forall x \in \mathbb{L}.F[x],\end{aligned} \tag{IndAxList}$$

for arbitrary first-order logic formulas F.

We use the induction axiom for lists (IndAxList) with formulas of the format (6.2). After preprocessing the *magic axioms for natural lists* then looks like

$$\begin{aligned}\mathsf{cnf}((G[\mathsf{nil}, u_0] \ \wedge (G[\sigma_l, \sigma_v] \to G[\mathsf{cons}(\sigma_n, \sigma_l), u_\mathsf{s}])) \\ \to G[x, \mathsf{rec}(u_\mathsf{nil}, u_\mathsf{cons}, x)]).\end{aligned} \tag{MagAxList}$$

The *recursive function over natural lists* that is *uniquely defined* by the pair $(u_\mathsf{nil}, u_\mathsf{cons}(.))$ with input x can then be written as

$$\begin{aligned}f(\mathsf{nil}) &= u_\mathsf{nil} \\ f(\mathsf{cons}(n, l)) &= u_\mathsf{cons}[f(l)].\end{aligned} \tag{6.4}$$

Remark 6.3.1 We note that the function in (6.4) does not depend on the variable n but only on the remaining part of the list l.

6.3.2 Natural Binary Trees

The inductive structure of binary trees consists of the constructors $\mathsf{leaf} : \mathbb{N} \to \mathbb{BT}$ of arity 1 and $\mathsf{bt} : \mathbb{N} \times \mathbb{BT} \times \mathbb{N} \to \mathbb{BT}$ of arity 3. Therefore the *structural induction axiom over natural binary trees* looks like

$$\begin{aligned}((\forall a \in \mathbb{N}.F[\mathsf{leaf}(a)]) \land \forall n \in \mathbb{N}, l, r \in \mathbb{BT}.((F[l] \land F[r]) \to F[\mathsf{bt}(l,n,r)])) \\ \to \forall x \in \mathbb{BT}.F[x],\end{aligned} \tag{IndAxTree}$$

for an arbitrary first-order logic formula F.

We, again, use the induction axiom (IndAxTree) together with a formula of the format (6.2). After correct preprocessing this results in the *magic axiom for natural binary trees*

$$\begin{aligned}(G[\mathsf{leaf}(a), u_0] \ \land ((G[\sigma_l, \sigma_v] \land G[\sigma_r, \sigma_v']) \to G[\mathsf{bt}(\sigma_l, \sigma_n, \sigma_r), u_\mathsf{s}]) \\ \to G[x, \mathsf{rec}(u_\mathsf{leaf}, u_\mathsf{bt}, x)].\end{aligned} \tag{MagAxTree}$$

The *recursive function over natural binary trees* that is *uniquely defined* by the pair $(u_\mathsf{leaf}, u_\mathsf{bt}(.,.))$ with input x can then be written as

$$\begin{aligned}f(\mathsf{leaf}(n)) &= u_\mathsf{leaf} \\ f(\mathsf{bt}(l,n,r)) &= u_\mathsf{bt}[f(l), f(r)].\end{aligned} \tag{6.5}$$

Remark 6.3.2 Note here, that this time the step case of the recursive function in (6.5) depends on two arguments, the left part of the tree, l, and the right part of the tree r.

7 Exploring Recursive Synthesis in Saturation over Different Inductive Structures

In this chapter, I want to showcase some of the examples I developed and realized while working with Laura Kovács and Petra Hozzová. The outcome of this work was depicted in the paper "Synthesis of Recursive Programs in Saturation", [HAH+24]. My contributions are presented in the Appendix D section of this very paper, which follows a similar structure to this chapter. Furthermore, all the tables, figures and program derivations are taken from there. The Appendix D section was written by myself, the formatting of Table 7.1 and proofreading was done by Petra Hozzová and Laura Kovács.

The lemmas used for the derivations are given in Figure 7.3. Note that contrary to the distributivity example in Chapter 5, the proofs of the lemmas are not written down explicitly, but can be formally proven like Lemma 4.2.2 with the use of the superposition calculus.

To highlight the important derivation steps, some literals and terms are typeset in bold. This is done either to mark on which literal induction is applied on or when an inference rule applied on the literal causes a rec-term to change.

Note that for examples using the predicate symbol "$<$", the expression $x \leq y$ is sometimes used as a syntactic macro for $\neg(y < x)$ to increase readability.

7.1 Recursive Synthesis over Natural Numbers

Example 7.1.1 (Subtraction with condition). This first example shows how two applications of induction result in a program consisting of two nested recursive functions. The goal is to synthesize the subtraction function. Due to the fact that we are working with natural numbers, we add a precondition in the specification that makes sure that the number that is subtracted is strictly smaller than the number that is subtracted from. This then ensures validity of the specification.

Specification.

$$\forall x_1, x_2 \in \mathbb{N}.\exists y \in \mathbb{N}.(x_2 < x_1 \rightarrow x_2 + y = x_1)$$

Here we are using the strictly smaller ordering, defined in Figure 7.1, that fulfills the axioms ($A7_{\mathbb{N}}$) and ($A8_{\mathbb{N}}$). For synthesizing the subtraction function, we apply induction two times, which will therefore lead to two nested recursive functions.

	Natural numbers $\mathbb{N}$	Natural lists $\mathbb{L}$	Natural binary trees $\mathbb{BT}$
Constructors	$0 : \mathbb{N}$ $\mathsf{s} : \mathbb{N} \to \mathbb{N}$	$\mathsf{nil} : \mathbb{L}$ $\mathsf{cons} : \mathbb{N} \times \mathbb{L} \to \mathbb{L}$	$\mathsf{leaf} : \mathbb{N} \to \mathbb{BT}$ $\mathsf{bt} : \mathbb{BT} \times \mathbb{N} \times \mathbb{BT} \to \mathbb{BT}$
Symbols	$+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ $\cdot : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ $< : \mathbb{N} \times \mathbb{N} \to \{\top, \bot\}$	$+\!\!+ : \mathbb{L} \times \mathbb{L} \to \mathbb{L}$ $\mathsf{len} : \mathbb{L} \to \mathbb{N}$ $\mathsf{suff} : \mathbb{L} \times \mathbb{L} \to \{\top, \bot\}$ $\mathsf{in}_{\mathbb{L}} : \mathbb{N} \times \mathbb{L} \to \{\top, \bot\}$	$\mathsf{in}_{\mathbb{BT}} : \mathbb{N} \times \mathbb{BT} \to \{\top, \bot\}$

Figure 7.1: Definitions for the theories of natural numbers $\mathbb{N}$, lists $\mathbb{L}$, and binary trees $\mathbb{BT}$. The axioms defining function and predicate symbols can be found in Figure (7.2). When it is clear from the context we will just write in instead of $\mathsf{in}_{\mathbb{L}}/\mathsf{in}_{\mathbb{BT}}$.

Preprocessing. Preprocessing results in the following two clauses

$$\sigma_2 < \sigma_1 \vee \mathsf{ans}(y) \tag{Inp1}$$
$$\sigma_2 + y \neq \sigma_1 \vee \mathsf{ans}(y), \tag{Inp2}$$

where σ_1 and σ_2 denote the skolem constants of variables x_1 and x_2 respectively.

Details of the magic. We firstly use the magic axiom with the formula $G[t, x] := L_1[t] \vee L_2[t, x]$, where $L_1[t] := \neg(t < \sigma_1)$ and $L_2[t, x] := t + x = \sigma_1$, hence we get

$$\begin{aligned}(G[0, u_0] \ &\wedge (G[\sigma_y, \sigma_w] \to G[\mathsf{s}(\sigma_y), u_\mathsf{s}])) \\ &\to G[x, \mathsf{rec}(u_0, u_\mathsf{s}, x)].\end{aligned} \tag{7.1}$$

When transforming (7.1) into CNF we end up with the following six clauses:

$$\neg L_1[0] \vee L_1[\sigma_y] \vee L_2[\sigma_y, \sigma_w] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_0, u_\mathsf{s}, z)] \tag{7.2}$$
$$\neg L_2[0, u_0] \vee L_1[\sigma_y] \vee L_2[\sigma_y, \sigma_w] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_0, u_\mathsf{s}, z)] \tag{7.3}$$
$$\neg L_1[0] \vee \neg L_1[\mathsf{s}(\sigma_y)] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_0, u_\mathsf{s}, z)] \tag{7.4}$$
$$\neg L_1[0] \vee \neg L_2[\mathsf{s}(\sigma_y), u_\mathsf{s}] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_0, u_\mathsf{s}, z)] \tag{7.5}$$
$$\neg L_2[0, u_0] \vee \neg L_1[\mathsf{s}(\sigma_y)] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_0, u_\mathsf{s}, z)] \tag{7.6}$$
$$\neg L_2[0, u_0] \vee \neg L_2[\mathsf{s}(\sigma_y), u_\mathsf{s}] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_0, u_\mathsf{s}, z)] \tag{7.7}$$

In our derivation we will only use the clauses (7.3), (7.6) and (7.7) since the literal $\neg L_1[0] : 0 < \sigma_1$ cannot be resolved.

The second application of induction uses the unit literal $L[t, x] : t = \mathsf{s}(x)$. The formula $\forall x \exists y. x = \mathsf{s}(y)$ does not hold; a contradiction can be easily deduced using the axiom $\forall x. 0 \neq \mathsf{s}(x)$. However, what can be proven is the statement $\forall x. \exists y. x \neq 0 \to x = \mathsf{s}(y)$. In order to prove this we use induction in a slightly different way; we start with $\mathsf{s}(0)$ as a base case.

$$\Big(\exists u_0. L[\mathsf{s}(0), u_0] \wedge \forall y. \big(y \neq 0 \wedge \exists w. L[y, w] \to \exists u_\mathsf{s}. L[\mathsf{s}(y), u_\mathsf{s}]\big)\Big) \to \forall x. \exists y. (x \neq 0 \to L[x, y]).$$

Note that using this induction axiom results in a different scheme for the synthesized recursive function: the base case of the function will be $\mathsf{s}(0)$, and the recursive case will be $\mathsf{s}(n)$ conditioned on $n \neq 0$. The CNF of the corresponding induction axiom for synthesis is:

$$\neg L[\mathsf{s}(0), u_0] \vee \sigma_y \neq 0 \vee x = 0 \vee L[x, \mathsf{rec}(u_0, u_\mathsf{s}, x)] \tag{7.8}$$
$$\neg L[\mathsf{s}(0), u_0] \vee L[\sigma_y, \sigma_w] \vee x = 0 \vee L[x, \mathsf{rec}(u_0, u_\mathsf{s}, x)] \tag{7.9}$$
$$\neg L[\mathsf{s}(0), u_0] \vee \neg L[\mathsf{s}(\sigma_y), u_\mathsf{s}] \vee x = 0 \vee L[x, \mathsf{rec}(u_0, u_\mathsf{s}, x)] \tag{7.10}$$

Derivation and program. Putting this alltogether we obtain the following program derivation. For readability, we denote the two rec-symbols used in the two magic formulas by $\mathsf{rec}_{\mathsf{sub}}$ and $\mathsf{rec}_{\mathsf{pre}}$.

1. $\boldsymbol{\sigma_2 < \sigma_1} \vee \mathsf{ans}(y)$ [Inp1]
2. $\boldsymbol{\sigma_2 + y \neq \sigma_1} \vee \mathsf{ans}(y)$ [Inp2]
3. $\boldsymbol{0 + u_0 \neq \sigma_1} \vee \neg(\sigma_y < \sigma_1) \vee \sigma_y + \sigma_w = \sigma_1 \vee \neg(\sigma_2 < \sigma_1) \vee$ $\vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(u_0, u_\mathsf{s}, \boldsymbol{\sigma_2}))$ [BR 2, (7.3)]
4. $\boldsymbol{0 + u_0 \neq \sigma_1} \vee \mathsf{s}(\sigma_y) < \sigma_1 \vee \neg(\sigma_2 < \sigma_1) \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(u_0, u_\mathsf{s}, \boldsymbol{\sigma_2}))$ [BR 2, (7.6)]
5. $\boldsymbol{0 + u_0 \neq \sigma_1} \vee \mathsf{s}(\sigma_y) + u_\mathsf{s} \neq \sigma_1 \vee \neg(\sigma_2 < \sigma_1) \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(u_0, u_\mathsf{s}, \boldsymbol{\sigma_2}))$ [BR 2, (7.7)]
6. $0 + u_0 \neq \sigma_1 \vee \neg(\sigma_y < \sigma_1) \vee \sigma_y + \sigma_w = \sigma_1 \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(u_0, u_\mathsf{s}, \sigma_2))$ [BR 1, 3]
7. $0 + u_0 \neq \sigma_1 \vee \mathsf{s}(\sigma_y) < \sigma_1 \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(u_0, u_\mathsf{s}, \sigma_2))$ [BR 1, 4]
8. $0 + u_0 \neq \sigma_1 \vee \mathsf{s}(\sigma_y) + u_\mathsf{s} \neq \sigma_1 \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(u_0, u_\mathsf{s}, \sigma_2))$ [BR 1, 5]
9. $\neg(\sigma_y < \sigma_1) \vee \sigma_y + \sigma_w = \sigma_1 \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\boldsymbol{\sigma_1}, u_\mathsf{s}, \sigma_2))$ [Sup, ER ($\mathrm{L1}_\mathbb{N}$), 6]
10. $\mathsf{s}(\sigma_y) < \sigma_1 \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\boldsymbol{\sigma_1}, u_\mathsf{s}, \sigma_2))$ [Sup, ER ($\mathrm{L1}_\mathbb{N}$), 7]
11. $\mathsf{s}(\sigma_y) + u_\mathsf{s} \neq \sigma_1 \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\boldsymbol{\sigma_1}, u_\mathsf{s}, \sigma_2))$ [Sup, ER ($\mathrm{L1}_\mathbb{N}$), 8]
12. $\sigma_y + \mathsf{s}(u_\mathsf{s}) \neq \sigma_1 \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, u_\mathsf{s}, \sigma_2))$ [Sup ($\mathrm{L2}_\mathbb{N}$), 11]
13. $\sigma_y < \sigma_1 \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, u_\mathsf{s}, \sigma_2))$ [BR ($\mathrm{L5}_\mathbb{N}$), 10]
14. $\sigma_y + \sigma_w = \sigma_1 \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, u_\mathsf{s}, \sigma_2))$ [BR 9, 13]
15. $\sigma_y + \mathsf{s}(u_\mathsf{s}) \neq \sigma_y + \sigma_w \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, u_\mathsf{s}, \sigma_2))$ [Sup 12, 14]
16. $\boldsymbol{\sigma_w \neq \mathsf{s}(u_\mathsf{s})} \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, u_\mathsf{s}, \sigma_2))$ [BR ($\mathrm{L14}_\mathbb{N}$), 15]
17. $\boldsymbol{\mathsf{s}(0) \neq \mathsf{s}(u_0)} \vee \sigma_y = \mathsf{s}(\sigma'_w) \vee \sigma_w = 0 \vee$ $\vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, \mathsf{rec}_{\mathsf{pre}}(u_0, u_\mathsf{s}, \boldsymbol{\sigma_w}), \sigma_2))$ [BR 16, (7.9)]
18. $\boldsymbol{\mathsf{s}(0) \neq \mathsf{s}(u_0)} \vee \mathsf{s}(\sigma_y) \neq \mathsf{s}(u_\mathsf{s}) \vee \sigma_w = 0 \vee$ $\vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, \mathsf{rec}_{\mathsf{pre}}(u_0, u_\mathsf{s}, \boldsymbol{\sigma_w}), \sigma_2))$ [BR 16, (7.10)]

19. $\sigma_y = \mathsf{s}(\sigma'_w) \vee \sigma_w = 0 \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, \mathsf{rec}_{\mathsf{pre}}(\mathbf{0}, u_{\mathsf{s}}, \sigma_w), \sigma_2))$ [ER 17]

20. $\mathbf{s(\sigma_y) \not\approx s(u_s)} \vee \sigma_w = 0 \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, \mathsf{rec}_{\mathsf{pre}}(\mathbf{0}, u_{\mathsf{s}}, \sigma_w), \sigma_2))$ [ER 18]

21. $\mathbf{s(\sigma_y) = s(s(\sigma'_w))} \vee \sigma_w = 0 \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, \mathsf{rec}_{\mathsf{pre}}(0, u_{\mathsf{s}}, \sigma_w), \sigma_2))$ [BR ($\mathrm{A2}_{\mathbb{N}}$), 19]

22. $\sigma_w = 0 \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, \mathsf{rec}_{\mathsf{pre}}(0, \mathbf{s(\sigma'_w)}, \sigma_w), \sigma_2))$ [BR 20, 21, ER]

23. $\sigma_w \not\approx 0 \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, u_{\mathsf{s}}, \sigma_2))$ [BR ($\mathrm{L6}_{\mathbb{N}}$), 13, 14]

24. $\mathsf{ans}(\mathsf{rec}_{\mathsf{sub}}(\sigma_1, \mathsf{rec}_{\mathsf{pre}}(0, \mathsf{s}(\sigma'_w), \sigma_w), \sigma_2))$ [BR 22, 23, ER]

25. $\square$ [answer literal removal 24]

After completing the proof, we construct the program from the content of the answer literal in the following way. We start by constructing the recursive function pre from the innermost rec-term

$$\mathsf{rec}_{\mathsf{pre}}(0, \mathsf{s}(\sigma'_w), \sigma_w).$$

Due to the fact that the base case of the magic axiom was changed to $\mathsf{s}(0)$, this results in the following recursive function

$$\mathsf{pre}(\mathsf{s}(0)) = 0$$
$$n \not\approx 0 \to \mathsf{pre}(\mathsf{s}(n)) = \mathsf{s}(\mathsf{pre}(n)).$$

The second recursive function coming from the $\mathsf{rec}_{\mathsf{sub}}$-term calls upon the pre function in the base case. This leads to the following recursive function

$$\mathsf{sub}(0) = x_1$$
$$\mathsf{sub}(\mathsf{s}(n)) = \mathsf{pre}(\mathsf{sub}(n)).$$

The synthesized program is then

$$\mathsf{sub}(x_2).$$

Note that following this program the inner function pre never has 0 as an input. We see that the program is doing what was specified: subtracting 1 from x_1 x_2-times.

Example 7.1.2 (Floored square root). The next example aims to compute the rounded root of a natural number. We specify this using the following two constraints.

Specification.

$$\forall x \exists y.(y \cdot y \leq x \wedge x < \mathsf{s}(y) \cdot \mathsf{s}(y))$$

Details of the magic. For this example and for Example 7.1.3, the specification contains two unit clauses. In order to derive the synthesized program, we use a magic axiom with $G[t,x] := L_1[t,x] \wedge L_2[t,x]$. The CNF of its magic formula is:

$$\neg L_1[0,u_0] \vee \neg L_2[0,u_0] \vee L_1[\sigma_y,\sigma_w] \vee L_1[z,\mathsf{rec}(u_0,u_\mathsf{s},z)] \tag{7.11}$$
$$\neg L_1[0,u_0] \vee \neg L_2[0,u_0] \vee L_1[\sigma_y,\sigma_w] \vee L_2[z,\mathsf{rec}(u_0,u_\mathsf{s},z)] \tag{7.12}$$
$$\neg L_1[0,u_0] \vee \neg L_2[0,u_0] \vee L_2[\sigma_y,\sigma_w] \vee L_1[z,\mathsf{rec}(u_0,u_\mathsf{s},z)] \tag{7.13}$$
$$\neg L_1[0,u_0] \vee \neg L_2[0,u_0] \vee L_2[\sigma_y,\sigma_w] \vee L_2[z,\mathsf{rec}(u_0,u_\mathsf{s},z)] \tag{7.14}$$
$$\neg L_1[0,u_0] \vee \neg L_2[0,u_0] \vee \neg L_1[\mathsf{s}(\sigma_y),u_\mathsf{s}] \vee \neg L_2[\mathsf{s}(\sigma_y),u_\mathsf{s}] \vee L_1[z,\mathsf{rec}(u_0,u_\mathsf{s},z)] \tag{7.15}$$
$$\neg L_1[0,u_0] \vee \neg L_2[0,u_0] \vee \neg L_1[\mathsf{s}(\sigma_y),u_\mathsf{s}] \vee \neg L_2[\mathsf{s}(\sigma_y),u_\mathsf{s}] \vee L_2[z,\mathsf{rec}(u_0,u_\mathsf{s},z)] \tag{7.16}$$

Note that after we resolve the formulas above with a premise $\neg L_1[t,x] \vee \neg L_2[t,x] \vee C \vee \mathsf{ans}(r[x])$, we obtain only three clauses:

$$\neg L_1[0,u_0] \vee \neg L_2[0,u_0] \vee L_1[\sigma_y,\sigma_w] \vee C \vee \mathsf{ans}(r[\mathsf{rec}(u_0,u_\mathsf{s},t)]) \tag{7.17}$$
$$\neg L_1[0,u_0] \vee \neg L_2[0,u_0] \vee L_2[\sigma_y,\sigma_w] \vee C \vee \mathsf{ans}(r[\mathsf{rec}(u_0,u_\mathsf{s},t)]) \tag{7.18}$$
$$\neg L_1[0,u_0] \vee \neg L_2[0,u_0] \vee \neg L_1[\mathsf{s}(\sigma_y),u_\mathsf{s}] \vee \neg L_2[\mathsf{s}(\sigma_y),u_\mathsf{s}] \vee C \vee \mathsf{ans}(r[\mathsf{rec}(u_0,u_\mathsf{s},t)]) \tag{7.19}$$

For the derivation, we use the instances $L_1[t,y] : y \cdot y \leq t$ and $L_2[t,y] : t < \mathsf{s}(y) \cdot \mathsf{s}(y)$.

Derivation and program.

1. $\boldsymbol{\sigma_x < y \cdot y \vee \neg(\sigma_x < \mathsf{s}(y) \cdot \mathsf{s}(y))} \vee \mathsf{ans}(y)$ [input]
2. $0 < u_0 \cdot u_0 \vee \neg(0 < \mathsf{s}(u_0) \cdot \mathsf{s}(u_0)) \vee \neg(\sigma_y < \sigma_w \cdot \sigma_w) \vee$
   $\vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \boldsymbol{\sigma_x}))$ [(7.17)]
3. $0 < u_0 \cdot u_0 \vee \neg(0 < \mathsf{s}(u_0) \cdot \mathsf{s}(u_0)) \vee \sigma_y < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee$
   $\vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \boldsymbol{\sigma_x}))$ [(7.18)]
4. $0 < u_0 \cdot u_0 \vee \neg(0 < \mathsf{s}(u_0) \cdot \mathsf{s}(u_0)) \vee \mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot u_\mathsf{s} \vee$
   $\vee \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \mathsf{s}(u_\mathsf{s})) \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \boldsymbol{\sigma_x}))$ [(7.19)]
5. $\boldsymbol{\mathbf{0} \neq u_0 \cdot u_0} \vee \neg(0 < \mathsf{s}(u_0) \cdot \mathsf{s}(u_0)) \vee \neg(\sigma_y < \sigma_w \cdot \sigma_w) \vee$
   $\vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_x))$ [BR ($\mathrm{L7}_\mathbb{N}$), 2]
6. $\boldsymbol{\mathbf{0} \neq u_0 \cdot u_0} \vee \neg(0 < \mathsf{s}(u_0) \cdot \mathsf{s}(u_0)) \vee \sigma_y < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee$
   $\vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_x))$ [BR ($\mathrm{L7}_\mathbb{N}$), 3]
7. $\boldsymbol{\mathbf{0} \neq u_0 \cdot u_0} \vee \neg(0 < \mathsf{s}(u_0) \cdot \mathsf{s}(u_0)) \vee \mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot u_\mathsf{s} \vee$
   $\vee \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \mathsf{s}(u_\mathsf{s})) \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_x))$ [BR ($\mathrm{L7}_\mathbb{N}$), 4]
8. $\neg(0 < \mathsf{s}(0) \cdot \mathsf{s}(0)) \vee \neg(\sigma_y < \sigma_w \cdot \sigma_w) \vee$
   $\vee \mathsf{ans}(\mathsf{rec}(\mathbf{0}, u_\mathsf{s}, \sigma_x))$ [ER ($\mathrm{A5}_\mathbb{N}$), 5]
9. $\neg(0 < \mathsf{s}(0) \cdot \mathsf{s}(0)) \vee \sigma_y < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee$
   $\vee \mathsf{ans}(\mathsf{rec}(\mathbf{0}, u_\mathsf{s}, \sigma_x))$ [ER ($\mathrm{A5}_\mathbb{N}$), 6]

10. $\neg(0 < \mathsf{s}(0) \cdot \mathsf{s}(0)) \vee \mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot u_\mathsf{s} \vee$
$\vee \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \mathsf{s}(u_\mathsf{s})) \vee \mathsf{ans}(\mathsf{rec}(\mathbf{0}, u_\mathsf{s}, \sigma_x))$ [ER ($A5_\mathbb{N}$), 7]

11. $\neg(0 < \mathsf{s}(0) \cdot 0 + \mathsf{s}(0)) \vee \neg(\sigma_y < \sigma_w \cdot \sigma_w) \vee$
$\vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [ER ($A6_\mathbb{N}$), 8]

12. $\neg(0 < \mathsf{s}(0) \cdot 0 + \mathsf{s}(0)) \vee \sigma_y < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee$
$\vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [ER ($A6_\mathbb{N}$), 9]

13. $\neg(0 < \mathsf{s}(0) \cdot 0 + \mathsf{s}(0)) \vee \mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot u_\mathsf{s} \vee$
$\vee \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \mathsf{s}(u_\mathsf{s})) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [ER ($A6_\mathbb{N}$), 10]

14. $\neg(0 < 0 + \mathsf{s}(0)) \vee \neg(\sigma_y < \sigma_w \cdot \sigma_w) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [ER ($A5_\mathbb{N}$), 11]

15. $\neg(0 < 0 + \mathsf{s}(0)) \vee \sigma_y < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [ER ($A5_\mathbb{N}$), 12]

16. $\neg(0 < 0 + \mathsf{s}(0)) \vee \mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot u_\mathsf{s} \vee \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \mathsf{s}(u_\mathsf{s})) \vee$
$\vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [ER ($A5_\mathbb{N}$), 13]

17. $\neg(0 < \mathsf{s}(0)) \vee \neg(\sigma_y < \sigma_w \cdot \sigma_w) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [Sup ($L1_\mathbb{N}$), 14]

18. $\neg(0 < \mathsf{s}(0)) \vee \sigma_y < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [Sup ($L1_\mathbb{N}$), 15]

19. $\neg(0 < \mathsf{s}(0)) \vee \mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot u_\mathsf{s} \vee \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \mathsf{s}(u_\mathsf{s})) \vee$
$\vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [Sup ($L1_\mathbb{N}$), 16]

20. $\neg(\sigma_y < \sigma_w \cdot \sigma_w) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [BR ($A8_\mathbb{N}$), 17]

21. $\sigma_y < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [BR ($A8_\mathbb{N}$), 18]

22. $\mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot u_\mathsf{s} \vee \mathbf{\neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \mathsf{s}(u_\mathsf{s}))} \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [BR ($A8_\mathbb{N}$), 19]

23. $\mathbf{\mathsf{s}(\sigma_y) < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee \mathsf{s}(\sigma_y) = \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w)} \vee$
$\vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [BR ($L8_\mathbb{N}$), 21]

24. $\mathsf{s}(\sigma_y) < \sigma_w \cdot \sigma_w \vee \mathsf{s}(\sigma_y) = \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee \mathsf{ans}(\mathsf{rec}(0, \boldsymbol{\sigma_w}, \sigma_x))$ [BR 22, 23, ER]

25. $\neg(\mathsf{s}(\sigma_y) < \sigma_w \cdot \sigma_w) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [BR ($L9_\mathbb{N}$), 20]

26. $\mathbf{\mathsf{s}(\sigma_y) = \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w)} \vee \mathsf{ans}(\mathsf{rec}(0, \sigma_w, \sigma_x))$ [BR 24, 25]

27. $\mathbf{\mathsf{s}(\sigma_y) \neq u_\mathsf{s} \cdot u_\mathsf{s}} \vee \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \mathsf{s}(u_\mathsf{s})) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_x))$ [BR ($L7_\mathbb{N}$), 22]

28. $\mathsf{s}(\sigma_y) < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(\mathsf{s}(\sigma_w)) \cdot \mathsf{s}(\mathsf{s}(\sigma_w))) \vee$
$\vee \mathsf{ans}(\mathsf{rec}(0, \mathbf{\mathsf{s}(\sigma_w)}, \sigma_x))$ [BR 23, 27, ER]

29. $\mathsf{s}(\sigma_y) < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee$
$\vee \neg(\mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) < \mathsf{s}(\mathsf{s}(\sigma_w)) \cdot \mathsf{s}(\mathsf{s}(\sigma_w))) \vee$
$\vee \mathsf{ans}(\mathsf{rec}(0, \mathsf{s}(\sigma_w), \sigma_x))$ [Sup 23, 28]

30. $\mathsf{s}(\sigma_y) < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee$
$\vee \neg(\mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) < \mathsf{s}(\mathsf{s}(\sigma_w)) \cdot \mathsf{s}(\sigma_w) + \mathsf{s}(\mathsf{s}(\sigma_w))) \vee$
$\vee \mathsf{ans}(\mathsf{rec}(0, \mathsf{s}(\sigma_w), \sigma_x))$ [Sup ($A6_\mathbb{N}$), 29]

31. $\mathsf{s}(\sigma_y) < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee$
$\vee \neg(\mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) + \mathsf{s}(\sigma_w) + \mathsf{s}(\mathsf{s}(\sigma_w))) \vee$
$\vee \mathsf{ans}(\mathsf{rec}(0, \mathsf{s}(\sigma_w), \sigma_x))$ [Sup ($\mathrm{L4}_{\mathbb{N}}$), 30]

32. $\mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) + \mathsf{s}(\sigma_w)$ [BR ($\mathrm{A1}_{\mathbb{N}}$), ($\mathrm{L10}_{\mathbb{N}}$)]

33. $\mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) + \mathsf{s}(\sigma_w) < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) + \mathsf{s}(\sigma_w) + \mathsf{s}(\mathsf{s}(\sigma_w))$ [BR ($\mathrm{A1}_{\mathbb{N}}$), ($\mathrm{L10}_{\mathbb{N}}$)]

34. $\mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) + \mathsf{s}(\sigma_w) + \mathsf{s}(\mathsf{s}(\sigma_w))$ [BR ($\mathrm{A9}_{\mathbb{N}}$), 32, 33]

35. $\mathsf{s}(\sigma_y) < \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \vee \mathsf{ans}(\mathsf{rec}(0, \mathsf{s}(\sigma_w), \sigma_x))$ [BR 31, 34]

36. $\boldsymbol{\mathsf{s}(\sigma_y) \neq \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w)} \vee \mathsf{ans}(\mathsf{rec}(0, \mathsf{s}(\sigma_w), \sigma_x))$ [BR ($\mathrm{L7}_{\mathbb{N}}$), 35]

37. $\mathsf{ans}(\mathsf{rec}(0, \textbf{if } \mathsf{s}(\sigma_y) = \mathsf{s}(\sigma_w) \cdot \mathsf{s}(\sigma_w) \textbf{ then } \mathsf{s}(\sigma_w) \textbf{ else } \sigma_w, \sigma_x)$ [BR″ 26, 36]

38. $\square$ [answer literal removal 37]

Note that in step 37 we use the adapted binary resolution rule from Figure 6.1, which therefore introduces an $\mathsf{if}-\mathsf{then}-\mathsf{else}$ into the rec-term.

We can translate the content of the answer literal into

$$f(x),$$

where f is the recursive function defined as

$$\begin{aligned} f(0) &= 0 \\ f(\mathsf{s}(n)) &= \mathsf{if}\ \mathsf{s}(n) = \mathsf{s}(f(n)) \cdot \mathsf{s}(f(n))\ \mathsf{then}\ \mathsf{s}(f(n))\ \mathsf{else}\ f(n). \end{aligned}$$

Example 7.1.3 (Floored divison). The next example is similar to the previous one, the goal is to synthesize the rounded division of two inputs. In order to preserve validity, we add a condition that prevents division by zero.

Specification.

$$\forall x_1, x_2 \exists y.(x_2 \neq 0 \rightarrow (y \cdot x_2 \leq x_1 \wedge x_1 < \mathsf{s}(y) \cdot x_2))$$

Details of the magic. The formula used with the magic axiom has the same structure as in Example 7.1.2. This time the specific literals are $L_1[t, y] : y \cdot \sigma_2 \leq t$ and $L_2[t, y] : t < \mathsf{s}(y) \cdot \sigma_2$.

Derivation and program.

1. $\sigma_2 \neq 0 \vee \mathsf{ans}(y)$ [input]

2. $\boldsymbol{\sigma_1 < y \cdot \sigma_2} \vee \boldsymbol{\neg(\sigma_1 < \mathsf{s}(y) \cdot \sigma_2)} \vee \mathsf{ans}(y)$ [input]

3. $0 < u_0 \cdot \sigma_2 \vee \neg(0 < \mathsf{s}(u_0) \cdot \sigma_2) \vee \neg(\sigma_y < \sigma_w \cdot \sigma_2) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \boldsymbol{\sigma_1}))$ [(7.17)]

4. $0 < u_0 \cdot \sigma_2 \vee \neg(0 < \mathsf{s}(u_0) \cdot \sigma_2) \vee \sigma_y < \mathsf{s}(\sigma_w) \cdot \sigma_2 \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \boldsymbol{\sigma_1}))$ [(7.18)]

5. $0 < u_0 \cdot \sigma_2 \vee \neg(0 < \mathsf{s}(u_0) \cdot \sigma_2) \vee \mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot \sigma_2 \vee$
$\vee\ \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \boldsymbol{\sigma_1}))$ [(7.19)]

6. $\mathbf{0 \neq u_0 \cdot \sigma_2} \vee \neg(0 < \mathsf{s}(u_0) \cdot \sigma_2) \vee \neg(\sigma_y < \sigma_w \cdot \sigma_2) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_1))$ [BR ($\mathrm{L7}_\mathbb{N}$), 3]

7. $\mathbf{0 \neq u_0 \cdot \sigma_2} \vee \neg(0 < \mathsf{s}(u_0) \cdot \sigma_2) \vee \sigma_y < \mathsf{s}(\sigma_w) \cdot \sigma_2 \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_1))$ [BR ($\mathrm{L7}_\mathbb{N}$), 4]

8. $\mathbf{0 \neq u_0 \cdot \sigma_2} \vee \neg(0 < \mathsf{s}(u_0) \cdot \sigma_2) \vee \mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot \sigma_2 \vee$
$\vee\ \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma_1))$ [BR ($\mathrm{L7}_\mathbb{N}$), 5]

9. $\neg(0 < \mathsf{s}(0) \cdot \sigma_2) \vee \neg(\sigma_y < \sigma_w \cdot \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(\mathbf{0}, u_\mathsf{s}, \sigma_1))$ [ER ($\mathrm{L3}_\mathbb{N}$), 6]

10. $\neg(0 < \mathsf{s}(0) \cdot \sigma_2) \vee \sigma_y < \mathsf{s}(\sigma_w) \cdot \sigma_2 \vee \mathsf{ans}(\mathsf{rec}(\mathbf{0}, u_\mathsf{s}, \sigma_1))$ [ER ($\mathrm{L3}_\mathbb{N}$), 7]

11. $\neg(0 < \mathsf{s}(0) \cdot \sigma_2) \vee \mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot \sigma_2 \vee$
$\vee\ \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(\mathbf{0}, u_\mathsf{s}, \sigma_1))$ [ER ($\mathrm{L3}_\mathbb{N}$), 8]

12. $\neg(0 < 0 \cdot \sigma_2 + \sigma_2) \vee \neg(\sigma_y < \sigma_w \cdot \sigma_2) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_1))$ [ER ($\mathrm{L4}_\mathbb{N}$), 9]

13. $\neg(0 < 0 \cdot \sigma_2 + \sigma_2) \vee \sigma_y < \mathsf{s}(\sigma_w) \cdot \sigma_2 \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_1))$ [ER ($\mathrm{L4}_\mathbb{N}$), 10]

14. $\neg(0 < 0 \cdot \sigma_2 + \sigma_2) \vee \mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot \sigma_2 \vee$
$\vee\ \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_1))$ [ER ($\mathrm{L4}_\mathbb{N}$), 11]

15. $\neg(0 < 0 + \sigma_2) \vee \neg(\sigma_y < \sigma_w \cdot \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_1))$ [Sup ($\mathrm{L3}_\mathbb{N}$), 12]

16. $\neg(0 < 0 + \sigma_2) \vee \sigma_y < \mathsf{s}(\sigma_w) \cdot \sigma_2 \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_1))$ [Sup ($\mathrm{L3}_\mathbb{N}$), 13]

17. $\neg(0 < 0 + \sigma_2) \vee \mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot \sigma_2 \vee \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_\mathsf{s}) \cdot \sigma_2) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_1))$ [Sup ($\mathrm{L3}_\mathbb{N}$), 14]

18. $x < x + \sigma_2 \vee \mathsf{ans}(y)$ [BR ($\mathrm{L10}_\mathbb{N}$), 1]

19. $\neg(\sigma_y < \sigma_w \cdot \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_1))$ [BR 15, 18]

20. $\sigma_y < \mathsf{s}(\sigma_w) \cdot \sigma_2 \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_1))$ [BR 16, 18]

21. $\mathsf{s}(\sigma_y) < u_\mathsf{s} \cdot \sigma_2 \vee \mathbf{\neg(s(\sigma_y) < s(u_s) \cdot \sigma_2)} \vee \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_1))$ [BR 17, 18]

22. $\mathbf{s(\sigma_y) < s(\sigma_w) \cdot \sigma_2} \vee \mathbf{s(\sigma_y) = s(\sigma_w) \cdot \sigma_2} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(0, u_\mathsf{s}, \sigma_1))$ [BR ($\mathrm{L8}_\mathbb{N}$), 20]

23. $\mathsf{s}(\sigma_y) < \sigma_w \cdot \sigma_2 \vee \mathsf{s}(\sigma_y) = \mathsf{s}(\sigma_w) \cdot \sigma_2 \vee$
$\vee\, \mathsf{ans}(\mathsf{rec}(0,\boldsymbol{\sigma_w}, \sigma_1))$ [BR 21, 22]

24. $\neg(\mathsf{s}(\sigma_y) < \sigma_w \cdot \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(0, u_{\mathsf{s}}, \sigma_1))$ [BR ($\mathrm{L9}_{\mathbb{N}}$), 19]

25. $\boldsymbol{\mathsf{s}(\sigma_y) = \mathsf{s}(\sigma_w) \cdot \sigma_2} \vee \mathsf{ans}(\mathsf{rec}(0, \sigma_w, \sigma_1))$ [BR 23, 24]

26. $\boldsymbol{\mathsf{s}(\sigma_y) \neq u_{\mathsf{s}} \cdot \sigma_2} \vee \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(u_{\mathsf{s}}) \cdot \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(0, u_{\mathsf{s}}, \sigma_1))$ [BR ($\mathrm{L7}_{\mathbb{N}}$), 21]

27. $\mathsf{s}(\sigma_y) < \mathsf{s}(\sigma_w) \cdot \sigma_2 \vee \neg(\mathsf{s}(\sigma_y) < \mathsf{s}(\mathsf{s}(\sigma_w)) \cdot \sigma_2) \vee$
$\vee\, \mathsf{ans}(\mathsf{rec}(0,\boldsymbol{\mathsf{s}(\sigma_w)}, \sigma_1))$ [BR 22, 26, ER]

28. $\mathsf{s}(\sigma_y) < \mathsf{s}(\sigma_w) \cdot \sigma_2 \vee \neg(\mathsf{s}(\sigma_w) \cdot \sigma_2 < \mathsf{s}(\mathsf{s}(\sigma_w)) \cdot \sigma_2) \vee$
$\vee\, \mathsf{ans}(\mathsf{rec}(0, \mathsf{s}(\sigma_w), \sigma_1))$ [Sup 22, 27]

29. $\mathsf{s}(\sigma_y) < \mathsf{s}(\sigma_w) \cdot \sigma_2 \vee \neg(\mathsf{s}(\sigma_w) \cdot \sigma_2 < \mathsf{s}(\sigma_w) \cdot \sigma_2 + \sigma_2) \vee$
$\vee\, \mathsf{ans}(\mathsf{rec}(0, \mathsf{s}(\sigma_w), \sigma_1))$ [Sup ($\mathrm{L4}_{\mathbb{N}}$), 28]

30. $\mathsf{s}(\sigma_y) < \mathsf{s}(\sigma_w) \cdot \sigma_2 \vee \mathsf{ans}(\mathsf{rec}(0, \mathsf{s}(\sigma_w), \sigma_1))$ [BR 18, 29]

31. $\boldsymbol{\mathsf{s}(\sigma_y) \neq \mathsf{s}(\sigma_w) \cdot \sigma_2} \vee \mathsf{ans}(\mathsf{rec}(0, \mathsf{s}(\sigma_w), \sigma_1))$ [BR ($\mathrm{L7}_{\mathbb{N}}$), 30]

32. $\mathsf{ans}(\mathsf{rec}(0,\boldsymbol{\mathsf{if}\ \mathsf{s}(\sigma_y) = \mathsf{s}(\sigma_w) \cdot \sigma_2\ \mathsf{then}\ \mathsf{s}(\sigma_w)\ \mathsf{else}\ \sigma_w}, \sigma_1))$ [BR″ 25, 31]

33. $\square$ [answer literal removal 32]

The program we obtain is

$$f(x_1),$$

where f is the recursive function defined as

$$\begin{aligned} f(0) &= 0 \\ f(\mathsf{s}(n)) &= \mathsf{if}\ \mathsf{s}(n) = \mathsf{s}(f(n)) \cdot x_2\ \mathsf{then}\ \mathsf{s}(f(n))\ \mathsf{else}\ f(n). \end{aligned}$$

$\square$

7.2 Recursive Synthesis over Natural Lists

Example 7.2.1 (Length of two concatenated lists). As the first example over lists, we want to compute the length of two concatenated lists.

Specification.

$$\forall x_1, x_2 \in \mathbb{L}.\exists y \in \mathbb{N}.y = \mathsf{len}(x_1 {+\!\!+} x_2)$$

To avoid the trivial solution of $\mathsf{len}(x_1 {+\!\!+} x_2)$, the symbol $+\!\!+$ is marked as uncomputable.

Details of the magic. This is our first example over the theory of lists. The structure of the magic axiom we use is similar to the one for natural numbers, just with different constructors. The 0-constructor changes to nil and the s-constructor to cons, with an additional argument n, which is in the magic formula skolemized as σ_n.

The CNF of the magic formula is:

$$\neg L[\mathsf{nil}, u_{\mathsf{nil}}] \vee L[\sigma_l, \sigma_w] \vee L[z, \mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, z)] \tag{7.20}$$

$$\neg L[\mathsf{nil}, u_{\mathsf{nil}}] \vee \neg L[\mathsf{cons}(\sigma_n, \sigma_l), u_{\mathsf{cons}}] \vee L[z, \mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, z)] \tag{7.21}$$

We will denote the application of induction with this magic formula by $\mathsf{MagInd}_{\mathbb{L}}$.

Derivation and program.

1. $\boldsymbol{y \not\approx \mathsf{len}(\sigma_1 +\!\!+ \sigma_2)} \vee \mathsf{ans}(y)$ [input]
2. $u_{\mathsf{nil}} \not\approx \mathsf{len}(\mathsf{nil} +\!\!+ \sigma_2) \vee \sigma_w \approx \mathsf{len}(\sigma_l +\!\!+ \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_1}))$ [$\mathsf{MagInd}_{\mathbb{L}}$, BR 1]
3. $u_{\mathsf{nil}} \not\approx \mathsf{len}(\mathsf{nil} +\!\!+ \sigma_2) \vee u_{\mathsf{cons}} \not\approx \mathsf{len}(\mathsf{cons}(\sigma_n, \sigma_l) +\!\!+ \sigma_2) \vee \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_1}))$ [$\mathsf{MagInd}_{\mathbb{L}}$, BR 1]
4. $\boldsymbol{u_{\mathsf{nil}} \not\approx \mathsf{len}(\sigma_2)} \vee \sigma_w \approx \mathsf{len}(\sigma_l +\!\!+ \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \sigma_1))$ [Sup ($\mathrm{L1}_{\mathbb{L}}$), 2]
5. $\sigma_w \approx \mathsf{len}(\sigma_l +\!\!+ \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\mathsf{len}(\sigma_2)}, u_{\mathsf{cons}}, \sigma_1))$ [ER 4]
6. $\boldsymbol{u_{\mathsf{nil}} \not\approx \mathsf{len}(\sigma_2)} \vee u_{\mathsf{cons}} \not\approx \mathsf{len}(\mathsf{cons}(\sigma_n, \sigma_l) +\!\!+ \sigma_2) \vee \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \sigma_1))$ [Sup ($\mathrm{L1}_{\mathbb{L}}$), 3]
7. $u_{\mathsf{cons}} \not\approx \mathsf{len}(\mathsf{cons}(\sigma_n, \sigma_l) +\!\!+ \sigma_2) \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\mathsf{len}(\sigma_2)}, u_{\mathsf{cons}}, \sigma_1))$ [ER 6]
8. $u_{\mathsf{cons}} \not\approx \mathsf{len}(\mathsf{cons}(\sigma_n, \sigma_l +\!\!+ \sigma_2)) \vee \mathsf{ans}(\mathsf{rec}(\mathsf{len}(\sigma_2), u_{\mathsf{cons}}, \sigma_1))$ [Sup ($\mathrm{A2}_{\mathbb{L}}$), 7]
9. $u_{\mathsf{cons}} \not\approx \mathsf{s}(\mathsf{len}(\sigma_l +\!\!+ \sigma_2)) \vee \mathsf{ans}(\mathsf{rec}(\mathsf{len}(\sigma_2), u_{\mathsf{cons}}, \sigma_1))$ [Sup ($\mathrm{A4}_{\mathbb{L}}$), 8]
10. $\boldsymbol{u_{\mathsf{cons}} \not\approx \mathsf{s}(\sigma_w)} \vee \mathsf{ans}(\mathsf{rec}(\mathsf{len}(\sigma_2), u_{\mathsf{cons}}, \sigma_1))$ [Sup 5, 9]
11. $\mathsf{ans}(\mathsf{rec}(\mathsf{len}(\sigma_2), \boldsymbol{\mathsf{s}(\sigma_w)}, \sigma_1))$ [ER 10]
12. $\square$ [answer literal removal 11]

The program constructed from $\mathsf{rec}(\mathsf{len}(x_2), \mathsf{s}(f(l)), x_1)$ is

$$f(x_1)$$

where

$$f(\mathsf{nil}) = \mathsf{len}(x_2)$$
$$f(\mathsf{cons}(n, l)) = \mathsf{s}(f(l)).$$

Example 7.2.2 (Last element of a list).

Specification. The specification is

$$\forall x \in \mathbb{L} \exists y \in \mathbb{N}.(x \neq \mathsf{nil} \rightarrow \exists z \in \mathbb{L}.x = z{+\!\!+}\mathsf{cons}(y, \mathsf{nil}))$$

.

Details of the magic. We apply induction with the base $\mathsf{cons}(a, \mathsf{nil})$, similarly how we did in Example 7.1.1. The CNF of the magic formula is:

$$\neg L[\mathsf{cons}(\sigma_a, \mathsf{nil}), u_{\mathsf{nil}}] \vee \sigma_l \neq \mathsf{nil} \vee z = \mathsf{nil} \vee L[z, \mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, z)] \tag{7.22}$$

$$\neg L[\mathsf{cons}(\sigma_a, \mathsf{nil}), u_{\mathsf{nil}}] \vee L[\sigma_l, \sigma_w] \vee z = \mathsf{nil} \vee L[z, \mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, z)] \tag{7.23}$$

$$\neg L[\mathsf{cons}(\sigma_a, \mathsf{nil}), u_{\mathsf{nil}}] \vee \neg L[\mathsf{s}(\sigma_l), u_{\mathsf{cons}}] \vee z = \mathsf{nil} \vee L[z, \mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, z)] \tag{7.24}$$

We will denote the application of induction with this magic formula by $\mathsf{MagInd}'_{\mathbb{L}}$.

For the derivation we instantiate the magic axiom with the formula $L[t, x] := \exists z \in \mathbb{L}.t = z{+\!\!+}\mathsf{cons}(x, \mathsf{nil})$.

Derivation and program.

1. $\sigma_x \neq \mathsf{nil} \vee \mathsf{ans}(y)$ [input]
2. $\boldsymbol{\sigma_x \neq z{+\!\!+}\mathsf{cons}(y, \mathsf{nil})} \vee \mathsf{ans}(y)$ [input]
3. $\mathsf{cons}(\sigma_a, \mathsf{nil}) \neq z_1{+\!\!+}\mathsf{cons}(u_{\mathsf{nil}}, \mathsf{nil}) \vee \sigma_l = \sigma_z{+\!\!+}\mathsf{cons}(\sigma_w, \mathsf{nil}) \vee$
   $\vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'_{\mathbb{L}}$, BR 2]
4. $\mathsf{cons}(\sigma_a, \mathsf{nil}) \neq z_1{+\!\!+}\mathsf{cons}(u_{\mathsf{nil}}, \mathsf{nil}) \vee$
   $\vee \mathsf{cons}(\sigma_n, \sigma_l) \neq z_2{+\!\!+}\mathsf{cons}(u_{\mathsf{cons}}, \mathsf{nil}) \vee$
   $\vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'_{\mathbb{L}}$, BR 2]
5. $\boldsymbol{\mathsf{cons}(\sigma_a, \mathsf{nil}) \neq z_1{+\!\!+}\mathsf{cons}(u_{\mathsf{nil}}, \mathsf{nil})} \vee \sigma_l = \sigma_z{+\!\!+}\mathsf{cons}(\sigma_w, \mathsf{nil}) \vee$
   $\vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \sigma_x))$ [BR 1, 3, ER]
6. $\boldsymbol{\mathsf{cons}(\sigma_a, \mathsf{nil}) \neq z_1{+\!\!+}\mathsf{cons}(u_{\mathsf{nil}}, \mathsf{nil})} \vee$
   $\vee \mathsf{cons}(\sigma_n, \sigma_l) \neq z_2{+\!\!+}\mathsf{cons}(u_{\mathsf{cons}}, \mathsf{nil}) \vee$
   $\vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \sigma_x))$ [BR 1, 4, ER]
7. $\sigma_l = \sigma_z{+\!\!+}\mathsf{cons}(\sigma_w, \mathsf{nil}) \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{cons}}, \sigma_x))$ [ER ($\mathrm{L1}_{\mathbb{L}}$), 5]
8. $\mathsf{cons}(\sigma_n, \sigma_l) \neq z_2{+\!\!+}\mathsf{cons}(u_{\mathsf{cons}}, \mathsf{nil}) \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{cons}}, \sigma_x))$ [ER ($\mathrm{L1}_{\mathbb{L}}$), 6]
9. $\boldsymbol{\mathsf{cons}(\sigma_n, \sigma_z{+\!\!+}\mathsf{cons}(\sigma_w, \mathsf{nil})) \neq z_2{+\!\!+}\mathsf{cons}(u_{\mathsf{cons}}, \mathsf{nil})} \vee$
   $\vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [Sup 7, 8]
10. $\mathsf{ans}(\mathsf{rec}(\sigma_a, \boldsymbol{\sigma_w}, \sigma_x))$ [Sup ($\mathrm{A2}_{\mathbb{L}}$), 9, ER]
11. $\square$ [answer literal removal 10]

We derive the program

$$f(x),$$

where f is the recursive function defined as

$$\begin{aligned} f(\mathsf{cons}(a, \mathsf{nil})) &= a \\ f(\mathsf{cons}(n, l)) &= f(l). \end{aligned}$$

Note. Similarly to Example 7.2.2, we could synthesize the function that returns the first element of a list. However, in practice such a negated skolemized specification would be instantly resolved using the destructor for cons.

Example 7.2.3 (Prefix of a list given its suffix). For the next example, we want to return the remaining part of a list x_1 that has the list x_2 as a suffix. The goal is to construct a function that removes the list x_2 of list x_1. Using the predicate symbol suff for checking whether list x_2 is the first part of list x_1 (see Figure 7.1 and Figure 7.3), we get the following specification.

Specification.

$$\forall x_1, x_2 \in \mathbb{L}. \exists y \in \mathbb{L}. (\mathsf{suff}(x_2, x_1) \to x_1 = y \mathbin{+\!\!+} x_2)$$

Note here, that we can see this as the analogue to Example 7.1.1 for lists; subtracting list x_2 from list x_1. This results in a very similar program derivation and constructed recursive function; two applications of induction and hence, two nested recursive functions.

Details of the magic. We need to apply induction on $\mathsf{suff}(\sigma_2, \sigma_1)$ and $\sigma_1 \neq y \mathbin{+\!\!+} \sigma_2$. Therefore we instantiate the magic axiom for lists with a disjunction $G[t, x] := L_1[t] \vee L_2[t, x]$. We obtain a formula analogous to the one from Example 7.1.1. The CNF of the corresponding magic formula is:

$$\neg L_1[\mathsf{nil}] \vee L_1[\sigma_l] \vee L_2[\sigma_l, \sigma_w] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, z)] \tag{7.25}$$
$$\neg L_2[\mathsf{nil}, u_{\mathsf{nil}}] \vee L_1[\sigma_l] \vee L_2[\sigma_l, \sigma_w] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, z)] \tag{7.26}$$
$$\neg L_1[\mathsf{nil}] \vee \neg L_1[\mathsf{cons}(\sigma_n, \sigma_l)] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, z)] \tag{7.27}$$
$$\neg L_1[\mathsf{nil}] \vee \neg L_2[\mathsf{cons}(\sigma_n, \sigma_l), u_{\mathsf{cons}}] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, z)] \tag{7.28}$$
$$\neg L_2[\mathsf{nil}, u_{\mathsf{nil}}] \vee \neg L_1[\mathsf{cons}(\sigma_n, \sigma_l)] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, z)] \tag{7.29}$$
$$\neg L_2[\mathsf{nil}, u_{\mathsf{nil}}] \vee \neg L_2[\mathsf{cons}(\sigma_n, \sigma_l), u_{\mathsf{cons}}] \vee L_1[z] \vee L_2[z, \mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, z)] \tag{7.30}$$

We will denote the application of induction with this magic formula by $\mathsf{MagInd}''_{\mathbb{L}}$.

After applying $\mathsf{MagInd}''_{\mathbb{L}}$ with $G[t, y] : \neg(\mathsf{suff}(t, \sigma_1)) \vee \sigma_1 = y \mathbin{+\!\!+} t$ in the derivation, we need to apply induction again. Similarly to Example 7.1.1, the second time we need a non-standard base case: instead of nil, we use $\mathsf{cons}(a, \mathsf{nil})$. The magic axiom is:

$$\begin{aligned} \Big(\forall a. \exists u_{\mathsf{nil}}. G[\mathsf{cons}(a, \mathsf{nil}), u_{\mathsf{nil}}] \wedge \forall n, l. \big(l \neq \mathsf{nil} \wedge \exists w. G[l, w] \to \exists u_{\mathsf{cons}}. G[\mathsf{cons}(n, l), u_{\mathsf{cons}}]\big)\Big) \\ \to \forall z. \exists x. (z \neq \mathsf{nil} \to G[z, x]) \end{aligned} \tag{7.31}$$

We will denote the application of induction with this magic formula by $\mathsf{MagInd}'''_{\mathbb{L}}$. We instantiate it with $G[t,x] := \neg(\mathsf{suff}(\mathsf{cons}(\sigma_n, \mathsf{nil}), t)) \vee t = x{+\!\!+}\mathsf{cons}(\sigma_n, \mathsf{nil})$.

Derivation and program.

1. $\mathbf{suff(\sigma_2, \sigma_1)} \vee \mathsf{ans}(y)$ [input]
2. $\boldsymbol{\sigma_1 \neq y{+\!\!+}\sigma_2} \vee \mathsf{ans}(y)$ [input]
3. $\boldsymbol{\sigma_1 \neq u_{\mathsf{nil}}{+\!\!+}\mathsf{nil}} \vee \neg\mathsf{suff}(\sigma_l, \sigma_1) \vee \sigma_1 = \sigma_w{+\!\!+}\sigma_l \vee$
   $\vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_2}))$ [$\mathsf{MagInd}''_{\mathbb{L}}$, BR 1, 2]
4. $\boldsymbol{\sigma_1 \neq u_{\mathsf{nil}}{+\!\!+}\mathsf{nil}} \vee \mathsf{suff}(\mathsf{cons}(\sigma_n, \sigma_l), \sigma_1) \vee$
   $\vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_2}))$ [$\mathsf{MagInd}''_{\mathbb{L}}$, BR 1, 2]
5. $\boldsymbol{\sigma_1 \neq u_{\mathsf{nil}}{+\!\!+}\mathsf{nil}} \vee \sigma_1 \neq u_{\mathsf{cons}}{+\!\!+}\mathsf{cons}(\sigma_n, \sigma_l) \vee$
   $\vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_2}))$ [$\mathsf{MagInd}''_{\mathbb{L}}$, BR 1, 2]
6. $\neg\mathsf{suff}(\sigma_l, \sigma_1) \vee \sigma_1 = \sigma_w{+\!\!+}\sigma_l \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\boldsymbol{\sigma_1}, u_{\mathsf{cons}}, \sigma_2))$ [Sup, ER ($\mathrm{A1}_{\mathbb{L}}$), 3]
7. $\mathsf{suff}(\mathsf{cons}(\sigma_n, \sigma_l), \sigma_1) \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\boldsymbol{\sigma_1}, u_{\mathsf{cons}}, \sigma_2))$ [Sup, ER ($\mathrm{A1}_{\mathbb{L}}$), 4]
8. $\sigma_1 \neq u_{\mathsf{cons}}{+\!\!+}\mathsf{cons}(\sigma_n, \sigma_l) \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\boldsymbol{\sigma_1}, u_{\mathsf{cons}}, \sigma_2))$ [Sup, ER ($\mathrm{A1}_{\mathbb{L}}$), 5]
9. $\mathsf{suff}(\sigma_l, \sigma_1) \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, u_{\mathsf{cons}}, \sigma_2))$ [BR ($\mathrm{A8}_{\mathbb{L}}$), 7]
10. $\sigma_1 = \sigma_w{+\!\!+}\sigma_l \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, u_{\mathsf{cons}}, \sigma_2))$ [BR 6, 9]
11. $\sigma_w{+\!\!+}\sigma_l \neq u_{\mathsf{cons}}{+\!\!+}\mathsf{cons}(\sigma_n, \sigma_l) \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, u_{\mathsf{cons}}, \sigma_2))$ [Sup 8, 10]
12. $\sigma_w{+\!\!+}\sigma_l \neq u_{\mathsf{cons}}{+\!\!+}\mathsf{cons}(\sigma_n, \mathsf{nil}{+\!\!+}\sigma_l) \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, u_{\mathsf{cons}}, \sigma_2))$ [Sup ($\mathrm{A1}_{\mathbb{L}}$), 11]
13. $\sigma_w{+\!\!+}\sigma_l \neq u_{\mathsf{cons}}{+\!\!+}\mathsf{cons}(\sigma_n, \mathsf{nil}){+\!\!+}\sigma_l \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, u_{\mathsf{cons}}, \sigma_2))$ [Sup ($\mathrm{A2}_{\mathbb{L}}$), 12]
14. $\boldsymbol{\sigma_w \neq u_{\mathsf{cons}}{+\!\!+}\mathsf{cons}(\sigma_n, \mathsf{nil})} \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, u_{\mathsf{cons}}, \sigma_2))$ [BR ($\mathrm{L6}_{\mathbb{L}}$), 13]
15. $\mathbf{suff(cons(\sigma_n, nil), \sigma_w)} \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, u_{\mathsf{cons}}, \sigma_2))$ [BR ($\mathrm{L3}_{\mathbb{L}}$), 7, 10]
16. $\mathbf{cons(a, nil) \neq u_{\mathsf{nil}}{+\!\!+}cons(\sigma_n, nil)} \vee \neg\mathsf{suff}(\mathsf{cons}(\sigma_n, \mathsf{nil}), \sigma_l) \vee$
    $\vee \sigma_l = \sigma'_w{+\!\!+}\mathsf{cons}(\sigma_n, \mathsf{nil}) \vee \sigma_w = \mathsf{nil} \vee$
    $\vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_w}), \sigma_2))$ [$\mathsf{MagInd}'''_{\mathbb{L}}$, BR 14, 15]
17. $\mathbf{cons(a, nil) \neq u_{\mathsf{nil}}{+\!\!+}cons(\sigma_n, nil)} \vee$
    $\vee \mathsf{suff}(\mathsf{cons}(\sigma_n, \mathsf{nil}), \mathsf{cons}(\sigma'_n, \sigma_l)) \vee \sigma_w = \mathsf{nil} \vee$
    $\vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_w}), \sigma_2))$ [$\mathsf{MagInd}'''_{\mathbb{L}}$, BR 14, 15]
18. $\mathbf{cons(a, nil) \neq u_{\mathsf{nil}}{+\!\!+}cons(\sigma_n, nil)} \vee$
    $\vee \mathsf{cons}(\sigma'_n, \sigma_l) \neq u_{\mathsf{cons}}{+\!\!+}\mathsf{cons}(\sigma_n, \mathsf{nil}) \vee \sigma_w = \mathsf{nil} \vee$
    $\vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_w}), \sigma_2))$ [$\mathsf{MagInd}'''_{\mathbb{L}}$, BR 14, 15]
19. $\mathbf{cons(a, nil) \neq u_{\mathsf{nil}}{+\!\!+}cons(\sigma_n, nil)} \vee \sigma_l \neq \mathsf{nil} \vee \sigma_w = \mathsf{nil} \vee$
    $\vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_w}), \sigma_2))$ [$\mathsf{MagInd}'''_{\mathbb{L}}$, BR 14, 15]

20. $\neg\mathsf{suff}(\mathsf{cons}(\sigma_n, \mathsf{nil}), \sigma_l) \vee \sigma_l = \sigma'_w{+\!\!+}\mathsf{cons}(\sigma_n, \mathsf{nil}) \vee \sigma_w = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(\mathbf{nil}, u_{\mathsf{cons}}, \sigma_w), \sigma_2))$ [BR ($\mathrm{A1}_{\mathbb{L}}$), 16]

21. $\mathsf{suff}(\mathsf{cons}(\sigma_n, \mathsf{nil}), \mathsf{cons}(\sigma'_n, \sigma_l)) \vee \sigma_w = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(\mathbf{nil}, u_{\mathsf{cons}}, \sigma_w), \sigma_2))$ [BR ($\mathrm{A1}_{\mathbb{L}}$), 17]

22. $\mathsf{cons}(\sigma'_n, \sigma_l) \neq u_{\mathsf{cons}}{+\!\!+}\mathsf{cons}(\sigma_n, \mathsf{nil}) \vee \sigma_w = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(\mathbf{nil}, u_{\mathsf{cons}}, \sigma_w), \sigma_2))$ [BR ($\mathrm{A1}_{\mathbb{L}}$), 18]

23. $\sigma_l \neq \mathsf{nil} \vee \sigma_w = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(\mathbf{nil}, u_{\mathsf{cons}}, \sigma_w), \sigma_2))$ [BR ($\mathrm{A1}_{\mathbb{L}}$), 19]

24. $\mathsf{cons}(\sigma_n, \mathsf{nil}) = \mathsf{cons}(\sigma'_n, \sigma_l) \vee \mathsf{suff}(\mathsf{cons}(\sigma_n, \mathsf{nil}), \sigma_l) \vee$
$\vee\ \sigma_w = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(\mathsf{nil}, u_{\mathsf{cons}}, \sigma_w), \sigma_2))$ [BR ($\mathrm{L2}_{\mathbb{L}}$), 21]

25. $\sigma_l = \mathsf{nil} \vee \mathsf{suff}(\mathsf{cons}(\sigma_n, \mathsf{nil}), \sigma_l) \vee \sigma_w = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(\mathsf{nil}, u_{\mathsf{cons}}, \sigma_w), \sigma_2))$ [BR ($\mathrm{A9}_{\mathbb{L}}$), 24]

26. $\mathsf{suff}(\mathsf{cons}(\sigma_n, \mathsf{nil}), \sigma_l) \vee \sigma_w = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(\mathsf{nil}, u_{\mathsf{cons}}, \sigma_w), \sigma_2))$ [BR 23, 25]

27. $\sigma_l = \sigma'_w{+\!\!+}\mathsf{cons}(\sigma_n, \mathsf{nil}) \vee \sigma_w = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(\mathsf{nil}, u_{\mathsf{cons}}, \sigma_w), \sigma_2))$ [BR 20, 26]

28. $\mathbf{cons}(\boldsymbol{\sigma'_n}, \boldsymbol{\sigma'_w}{+\!\!+}\mathbf{cons}(\boldsymbol{\sigma_n}, \mathbf{nil})) \neq \boldsymbol{u_{\mathsf{cons}}}{+\!\!+}\mathbf{cons}(\boldsymbol{\sigma_n}, \mathbf{nil}) \vee$
$\vee\ \sigma_w = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(\mathsf{nil}, u_{\mathsf{cons}}, \sigma_w), \sigma_2))$ [Sup 22, 27]

29. $\sigma_w = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(\mathsf{nil}, \mathbf{cons}(\boldsymbol{\sigma'_n}, \boldsymbol{\sigma'_w}), \sigma_w), \sigma_2))$ [Sup ($\mathrm{A2}_{\mathbb{L}}$), 28]

30. $\sigma_1 \neq \sigma_l \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, u_{\mathsf{cons}}, \sigma_2))$ [BR ($\mathrm{L4}_{\mathbb{L}}$), 7]

31. $\sigma_1 = \sigma_l \vee \sigma_w \neq \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, u_{\mathsf{cons}}, \sigma_2))$ [BR ($\mathrm{L5}_{\mathbb{L}}$), 10]

32. $\sigma_w \neq \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, u_{\mathsf{cons}}, \sigma_2))$ [BR 30, 31]

33. $\mathsf{ans}(\mathsf{rec}_{\mathsf{pref}}(\sigma_1, \mathsf{rec}_{\mathsf{remove}}(\mathsf{nil}, \mathsf{cons}(\sigma'_n, \sigma'_w), \sigma_w), \sigma_2))$ [BR, ER 29, 32]

34. $\square$ [answer literal removal 33]

The program is constructed as

$$\mathsf{pref}(x_2),$$

where pref is the recursive function defined as

$$\begin{aligned}\mathsf{pref}(\mathsf{nil}) &= x_1\\ \mathsf{pref}(\mathsf{cons}(n, l)) &= \mathsf{remove}(\mathsf{pref}(l)),\end{aligned}$$

and remove is the recursive function defined as

$$\begin{aligned}\mathsf{remove}(\mathsf{cons}(a,\mathsf{nil})) &= \mathsf{nil}\\ \mathsf{remove}(\mathsf{cons}(n,l)) &= \mathsf{cons}(n,\mathsf{remove}(l)).\end{aligned}$$

Example 7.2.4 (Maximum element of a list). This next example aims to return the maximum element, given the well-founded ordering "$\leq$" of an arbitrary list as an input. The specification is then the following.

Specification.

$$\forall x \in \mathbb{L}\exists y \in \mathbb{N}.(x \neq \mathsf{nil} \rightarrow (\mathsf{in}(y,x) \wedge \forall k \in \mathbb{N}(\mathsf{in}(k,x) \rightarrow k \leq y)$$

The expression $\mathsf{in}(y,x)$ returns true, if the element y occurs in the list x. For a formal definition, see Figure 7.1 and Figure 7.3. The specification then verifies if the element y is a member of the input list x and if for every member of x it holds that $k \leq y$.

Details of the magic. The magic axiom is used with a formula of the form $G[t,x] := L_1[t,x] \wedge (L_2[t] \vee L_3[x])$. Similarly, as in the previous example, we choose $\mathsf{cons}(a,\mathsf{nil})$ for the base case. We will denote the application of induction with this magic axiom by $\mathsf{MagInd}'''''_{\mathbb{L}}$, and we apply it with $G[t,x] := \mathsf{in}(x,t) \wedge (\neg\mathsf{in}(\sigma_k,t) \vee \sigma_k \leq x)$.

Derivation and program.

1. $\sigma_x \neq \mathsf{nil} \vee \mathsf{ans}(y)$ [input]
2. $\mathbf{in(\sigma_k, \sigma_x)} \vee \mathbf{\neg in(y, \sigma_x)} \vee \mathsf{ans}(y)$ [input]
3. $\boldsymbol{y < \sigma_k} \vee \mathbf{\neg in(y, \sigma_x)} \vee \mathsf{ans}(y)$ [input]
4. $\mathbf{\neg in(u_{nil}, cons(\sigma_a, nil))} \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_a, \mathsf{nil})) \vee$
   $\vee\ \mathsf{in}(\sigma_w, \sigma_l) \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'''''_{\mathbb{L}}$, BR 2, 3]
5. $\mathbf{\neg in(u_{nil}, cons(\sigma_a, nil))} \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_a, \mathsf{nil})) \vee$
   $\vee\ \neg\mathsf{in}(\sigma_k, \sigma_l) \vee \sigma_k \leq \sigma_w \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'''''_{\mathbb{L}}$, BR 2, 3]
6. $\mathbf{\neg in(u_{nil}, cons(\sigma_a, nil))} \vee u_{\mathsf{nil}} < \sigma_k \vee \mathsf{in}(\sigma_w, \sigma_l) \vee$
   $\vee\ \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'''''_{\mathbb{L}}$, BR 2, 3]
7. $\mathbf{\neg in(u_{nil}, cons(\sigma_a, nil))} \vee u_{\mathsf{nil}} < \sigma_k \vee \neg\mathsf{in}(\sigma_k, \sigma_l) \vee$
   $\vee\ \sigma_k \leq \sigma_w \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'''''_{\mathbb{L}}$, BR 2, 3]
8. $\mathbf{\neg in(u_{nil}, cons(\sigma_a, nil))} \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_a, \mathsf{nil})) \vee$
   $\vee\ \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_n, \sigma_l)) \vee$
   $\vee\ \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'''''_{\mathbb{L}}$, BR 2, 3]
9. $\mathbf{\neg in(u_{nil}, cons(\sigma_a, nil))} \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_a, \mathsf{nil})) \vee$
   $\vee\ \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee u_{\mathsf{cons}} < \sigma_k \vee \sigma_x = \mathsf{nil} \vee$
   $\vee\ \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'''''_{\mathbb{L}}$, BR 2, 3]

10. $\neg\mathsf{in}(\boldsymbol{u_{\mathsf{nil}}}, \mathsf{cons}(\boldsymbol{\sigma_a}, \mathsf{nil})) \vee u_{\mathsf{nil}} < \sigma_k \vee$
$\vee \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_n, \sigma_l)) \vee$
$\vee\ \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}''''_{\mathbb{L}}$, BR 2, 3]

11. $\neg\mathsf{in}(\boldsymbol{u_{\mathsf{nil}}}, \mathsf{cons}(\boldsymbol{\sigma_a}, \mathsf{nil})) \vee u_{\mathsf{nil}} < \sigma_k \vee$
$\vee \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee u_{\mathsf{cons}} < \sigma_k \vee$
$\vee\ \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{nil}}, u_{\mathsf{cons}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}''''_{\mathbb{L}}$, BR 2, 3]

12. $\mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_a, \mathsf{nil})) \vee \mathsf{in}(\sigma_w, \sigma_l) \vee \sigma_x = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A10}_{\mathbb{L}}$), ($\mathrm{A11}_{\mathbb{L}}$), 4]

13. $\mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_a, \mathsf{nil})) \vee \neg\mathsf{in}(\sigma_k, \sigma_l) \vee \sigma_k \leq \sigma_w \vee \sigma_x = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A10}_{\mathbb{L}}$), ($\mathrm{A11}_{\mathbb{L}}$), 5]

14. $\sigma_a < \sigma_k \vee \mathsf{in}(\sigma_w, \sigma_l) \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A10}_{\mathbb{L}}$), ($\mathrm{A11}_{\mathbb{L}}$), 6]

15. $\sigma_a < \sigma_k \vee \neg\mathsf{in}(\sigma_k, \sigma_l) \vee \sigma_k \leq \sigma_w \vee \sigma_x = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A10}_{\mathbb{L}}$), ($\mathrm{A11}_{\mathbb{L}}$), 7]

16. $\mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_a, \mathsf{nil})) \vee \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_n, \sigma_l)) \vee$
$\vee\ \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A10}_{\mathbb{L}}$), ($\mathrm{A11}_{\mathbb{L}}$), 8]

17. $\mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_a, \mathsf{nil})) \vee \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee$
$\vee\ u_{\mathsf{cons}} < \sigma_k \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A10}_{\mathbb{L}}$), ($\mathrm{A11}_{\mathbb{L}}$), 9]

18. $\sigma_a < \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_n, \sigma_l)) \vee$
$\vee\ \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A10}_{\mathbb{L}}$), ($\mathrm{A11}_{\mathbb{L}}$), 10]

19. $\sigma_a < \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee u_{\mathsf{cons}} < \sigma_k \vee \sigma_x = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A10}_{\mathbb{L}}$), ($\mathrm{A11}_{\mathbb{L}}$), 11]

20. $\sigma_a \neq \sigma_k \vee \mathsf{in}(\sigma_w, \sigma_l) \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{L7}_{\mathbb{N}}$), 14]

21. $\sigma_a \neq \sigma_k \vee \neg\mathsf{in}(\sigma_k, \sigma_l) \vee \sigma_k \leq \sigma_w \vee \sigma_x = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{L7}_{\mathbb{N}}$), 15]

22. $\sigma_a \neq \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_n, \sigma_l)) \vee$
$\vee\ \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{L7}_{\mathbb{N}}$), 18]

23. $\sigma_a \neq \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee u_{\mathsf{cons}} < \sigma_k \vee \sigma_x = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{L7}_{\mathbb{N}}$), 19]

24. $\sigma_k = \sigma_a \vee \mathsf{in}(\sigma_w, \sigma_l) \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A11}_{\mathbb{L}}$), 12]

25. $\sigma_k = \sigma_a \vee \neg\mathsf{in}(\sigma_k, \sigma_l) \vee \sigma_k \leq \sigma_w \vee \sigma_x = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A11}_{\mathbb{L}}$), 13]

26. $\sigma_k = \sigma_a \vee \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee$
$\vee\ \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A11}_{\mathbb{L}}$), 16]

27. $\sigma_k = \sigma_a \vee \neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l))$
$\vee\ u_{\mathsf{cons}} < \sigma_k \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A11}_{\mathbb{L}}$), 17]

28. $\boldsymbol{\mathsf{in}(\sigma_w, \sigma_l)} \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR 20, 24]

29. $\neg\mathsf{in}(\sigma_k, \sigma_l) \vee \sigma_k \leq \sigma_w \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR 19, 25]

30. $\neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \sigma_x = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR 22, 26]

31. $\neg\mathsf{in}(u_{\mathsf{cons}}, \mathsf{cons}(\sigma_n, \sigma_l)) \vee u_{\mathsf{cons}} < \sigma_k \vee \sigma_x = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR 23, 27]

32. $\boldsymbol{u_{\mathsf{cons}} \neq \sigma_n} \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \sigma_x = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A11}_{\mathbb{L}}$), 30]

33. $\boldsymbol{\neg\mathsf{in}(u_{\mathsf{cons}}, \sigma_l)} \vee \mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \sigma_x = \mathsf{nil} \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A11}_{\mathbb{L}}$), 30]

34. $u_{\mathsf{cons}} \neq \sigma_n \vee u_{\mathsf{cons}} < \sigma_k \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A11}_{\mathbb{L}}$), 31]

35. $\neg\mathsf{in}(u_{\mathsf{cons}}, \sigma_l) \vee u_{\mathsf{cons}} < \sigma_k \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{cons}}, \sigma_x))$ [BR ($\mathrm{A11}_{\mathbb{L}}$), 30]

36. $\mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \boldsymbol{\sigma_w}, \sigma_x))$ [BR 28, 33, ER]

37. $\sigma_k = \sigma_n \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [BR ($\mathrm{A11}_{\mathbb{L}}$), 36]

38. $\sigma_w < \sigma_k \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [BR 28, 35]

39. $\neg\sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [BR 29, 38]

40. $\sigma_k = \sigma_n \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [BR 37, 39]

41. $\boldsymbol{\sigma_w < \sigma_n} \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [Sup 38, 40]

42. $\mathsf{in}(\sigma_k, \mathsf{cons}(\sigma_n, \sigma_l)) \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \boldsymbol{\sigma_n}, \sigma_x))$ [ER 32]

43. $\sigma_n < \sigma_k \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [ER 34]

44. $\sigma_k = \sigma_n \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR ($\mathrm{A11}_{\mathbb{L}}$), 42]

45. $\sigma_k = \sigma_n \vee \sigma_k \leq \sigma_w \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR 29, 44]

46. $\neg(\sigma_n < \sigma_k) \vee \sigma_k \leq \sigma_w \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR ($\mathrm{L7}_{\mathbb{N}}$), 45]

47. $\sigma_k \leq \sigma_w \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR 43, 46]

48. $\sigma_n \leq \sigma_k \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR ($\mathrm{L11}_{\mathbb{N}}$), 43]

49. $\boldsymbol{\sigma_n \leq \sigma_w} \vee \sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR ($\mathrm{A9}_{\mathbb{N}}$), 47, 48]

50. $\sigma_x = \mathsf{nil} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \boldsymbol{\mathsf{if}\ \sigma_w < \sigma_n\ \mathsf{then}\ \sigma_n\ \mathsf{else}\ \sigma_w}, \sigma_x))$ [BR′ 41, 49]

51. $\mathsf{ans}(\mathsf{rec}(\sigma_a, \mathsf{if}\ \sigma_w < \sigma_n\ \mathsf{then}\ \sigma_n\ \mathsf{else}\ \sigma_w, \sigma_x))$ [BR, ER 1, 50]
52. $\square$ [answer literal removal 51]

The program is then

$$f(x)$$

where f is the recursive function defined as

$$f(\mathsf{cons}(a, \mathsf{nil})) = a$$
$$f(\mathsf{cons}(n, l)) = \mathsf{if}\ f(l) < n\ \mathsf{then}\ n\ \mathsf{else}\ f(l).$$

7.3 Recursive Synthesis over Natural Binary Trees

Example 7.3.1 (Maximum element of a tree). This example is the analogous to Example 7.2.4 over binary trees. The intended meaning of the predicate symbol in is the same but defined over binary trees, see Figure 7.1 and Figure 7.3 for the formal definition. The specification is then the same as in Example 7.2.4 with the difference of using the predicate symbol in over binary trees. This results in a more complex derivation.

Specification.

$$\forall x \in \mathbb{BT} \exists y \in \mathbb{N}.(\mathsf{in}(y, x) \wedge \forall k \in \mathbb{N}(\mathsf{in}(k, x) \rightarrow k \leq y)$$

Details of the magic. The structure of the formula used to instantiate the magic axiom is the same as in Example 7.2.4, $G[t, x] := L_1[t, x] \wedge (L_2[t] \vee L_3[x])$, using the standard base case $\mathsf{leaf}(a)$. We will denote the application of induction with this magic axiom by $\mathsf{MagInd}'_{\mathbb{BT}}$, and we apply it with $G[t, x] := \mathsf{in}(x, t) \wedge (\neg\mathsf{in}(\sigma_k, t) \vee \sigma_k \leq x)$.

Derivation and program.

1. $\mathbf{in(\sigma_k, \sigma_x)} \vee \neg\mathbf{in(y, \sigma_x)} \vee \mathsf{ans}(y)$ [input]
2. $\boldsymbol{y < \sigma_k} \vee \neg\mathbf{in(y, \sigma_x)} \vee \mathsf{ans}(y)$ [input]
3. $\neg\mathbf{in}(u_{\mathsf{leaf}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee \mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee$
   $\vee\ \mathsf{in}(\sigma_w, \sigma_l) \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'_{\mathbb{BT}}$, BR 1, 2]
4. $\neg\mathbf{in}(u_{\mathsf{leaf}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee \mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee$
   $\vee\ \neg\mathsf{in}(k, \sigma_l) \vee \neg(\sigma_w < k) \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'_{\mathbb{BT}}$, BR 1, 2]
5. $\neg\mathbf{in}(u_{\mathsf{leaf}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee u_{\mathsf{leaf}} < \sigma_k \vee \mathsf{in}(\sigma_w, \sigma_l) \vee$
   $\vee\ \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'_{\mathbb{BT}}$, BR 1, 2]
6. $\neg\mathbf{in}(u_{\mathsf{leaf}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee u_{\mathsf{leaf}} < \sigma_k \vee \mathsf{in}(\sigma_w, \sigma_l) \vee$
   $\vee\ \sigma_w < \sigma_k \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'_{\mathbb{BT}}$, BR 1, 2]
7. $\neg\mathbf{in}(u_{\mathsf{leaf}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee \mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee$
   $\vee\ \mathsf{in}(\sigma'_w, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [$\mathsf{MagInd}'_{\mathbb{BT}}$, BR 1, 2]

8. $\neg\mathbf{in}(\boldsymbol{u_{\mathsf{leaf}}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee \mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee$
$\vee\ \neg\mathsf{in}(k, \sigma_r) \vee \neg(\sigma'_w < k) \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [MagInd$'_{\mathbb{BT}}$, BR 1, 2]

9. $\neg\mathbf{in}(\boldsymbol{u_{\mathsf{leaf}}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee u_{\mathsf{leaf}} < \sigma_k \vee \mathsf{in}(\sigma'_w, \sigma_r) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [MagInd$'_{\mathbb{BT}}$, BR 1, 2]

10. $\neg\mathbf{in}(\boldsymbol{u_{\mathsf{leaf}}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee u_{\mathsf{leaf}} < \sigma_k \vee \neg\mathsf{in}(k, \sigma_r) \vee$
$\vee\ \neg(\sigma'_w < k) \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [MagInd$'_{\mathbb{BT}}$, BR 1, 2]

11. $\neg\mathbf{in}(\boldsymbol{u_{\mathsf{leaf}}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee \mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee$
$\vee\ \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee \mathsf{in}(\sigma_k, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [MagInd$'_{\mathbb{BT}}$, BR 1, 2]

12. $\neg\mathbf{in}(\boldsymbol{u_{\mathsf{leaf}}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee \mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee$
$\vee\ u_{\mathsf{bt}} < \sigma_k \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [MagInd$'_{\mathbb{BT}}$, BR 1, 2]

13. $\neg\mathbf{in}(\boldsymbol{u_{\mathsf{leaf}}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee u_{\mathsf{leaf}} < \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee$
$\vee\ \mathsf{in}(\sigma_k, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [MagInd$'_{\mathbb{BT}}$, BR 1, 2]

14. $\neg\mathbf{in}(\boldsymbol{u_{\mathsf{leaf}}}, \mathbf{leaf}(\boldsymbol{\sigma_a})) \vee u_{\mathsf{leaf}} < \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee$
$\vee\ u_{\mathsf{bt}} < \sigma_k \vee \mathsf{ans}(\mathsf{rec}(u_{\mathsf{leaf}}, u_{\mathsf{bt}}, \boldsymbol{\sigma_x}))$ [MagInd$'_{\mathbb{BT}}$, BR 1, 2]

15. $\mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee \mathsf{in}(\sigma_w, \sigma_l) \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 3, ER]

16. $\mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee \neg\mathsf{in}(k, \sigma_l) \vee \neg(\sigma_w < k) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 4, ER]

17. $\sigma_a < \sigma_k \vee \mathsf{in}(\sigma_w, \sigma_l) \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 5, ER]

18. $\sigma_a < \sigma_k \vee \neg\mathsf{in}(k, \sigma_l) \vee \neg(\sigma_w < k) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 6, ER]

19. $\mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee \mathsf{in}(\sigma'_w, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 7, ER]

20. $\mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee \neg\mathsf{in}(k, \sigma_r) \vee \neg(\sigma'_w < k) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 8, ER]

21. $\sigma_a < \sigma_k \vee \mathsf{in}(\sigma'_w, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 9, ER]

22. $\sigma_a < \sigma_k \vee \neg\mathsf{in}(k, \sigma_r) \vee \neg(\sigma'_w < k) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 10, ER]

23. $\mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee$
$\vee\ \mathsf{in}(\sigma_k, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 11, ER]

24. $\mathsf{in}(\sigma_k, \mathsf{leaf}(\sigma_a)) \vee \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee$
$\vee\ u_{\mathsf{bt}} < \sigma_k \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 12, ER]

25. $\sigma_a < \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee$
$\vee\ \mathsf{in}(\sigma_k, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 13, ER]

26. $\sigma_a < \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee u_{\mathsf{bt}} < \sigma_k \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\boldsymbol{\sigma_a}, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 14, ER]

27. $\sigma_a = \sigma_k \vee \mathsf{in}(\sigma_w, \sigma_l) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 15]

28. $\sigma_a = \sigma_k \vee \neg\mathsf{in}(k, \sigma_l) \vee \neg(\sigma_w < k) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 16]

29. $\sigma_a \neq \sigma_k \vee \mathsf{in}(\sigma_w, \sigma_l) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{L12}_{\mathbb{N}}$), 17]

30. $\sigma_a \neq \sigma_k \vee \neg\mathsf{in}(k, \sigma_l) \vee \neg(\sigma_w < k) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{L12}_{\mathbb{N}}$), 18]

31. $\sigma_a = \sigma_k \vee \mathsf{in}(\sigma'_w, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 19]

32. $\sigma_a = \sigma_k \vee \neg\mathsf{in}(k, \sigma_r) \vee \neg(\sigma'_w < k) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 20]

33. $\sigma_a \neq \sigma_k \vee \mathsf{in}(\sigma'_w, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{L12}_{\mathbb{N}}$), 21]

34. $\sigma_a \neq \sigma_k \vee \neg\mathsf{in}(k, \sigma_r) \vee \neg(\sigma'_w < k) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{L12}_{\mathbb{N}}$), 22]

35. $\sigma_a = \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee$
$\vee\ \mathsf{in}(\sigma_k, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 23]

36. $\sigma_a = \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee u_{\mathsf{bt}} < \sigma_k \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A1}_{\mathbb{BT}}$), 24]

37. $\sigma_a \neq \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee$
$\vee\ \mathsf{in}(\sigma_k, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{L12}_{\mathbb{N}}$), 25]

38. $\sigma_a \neq \sigma_k \vee \neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee u_{\mathsf{bt}} < \sigma_k \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{L12}_{\mathbb{N}}$), 26]

39. $\boldsymbol{\mathsf{in}(\sigma_w, \sigma_l)} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR 27, 29]

40. $\neg\mathsf{in}(k, \sigma_l) \vee \neg(\sigma_w < k) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR 28, 30]

41. $\boldsymbol{\mathsf{in}(\sigma'_w, \sigma_r)} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR 31, 33]

42. $\neg\mathsf{in}(k, \sigma_r) \vee \neg(\sigma'_w < k) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR 32, 34]

43. $\neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee \mathsf{in}(\sigma_k, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR 35, 37]

44. $\neg\mathsf{in}(u_{\mathsf{bt}}, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee u_{\mathsf{bt}} < \sigma_k \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR 36, 38]

45. $u_{\mathsf{bt}} \neq \sigma_n \vee \mathsf{in}(\sigma_k, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A2}_{\mathbb{BT}}$), 43]

46. $\neg\mathsf{in}(u_{\mathsf{bt}}, \sigma_l) \vee \mathsf{in}(\sigma_k, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A2}_{\mathbb{BT}}$), 43]

47. $\neg\mathsf{in}(u_{\mathsf{bt}}, \sigma_r) \vee \mathsf{in}(\sigma_k, \mathsf{bt}(\sigma_l, \sigma_n, \sigma_r)) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A2}_{\mathbb{BT}}$), 43]

48. $\boldsymbol{u_{\mathsf{bt}} \neq \sigma_n} \vee \sigma_k = \sigma_n \vee \mathsf{in}(\sigma_k, \sigma_l) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee$
$\vee\ \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A2}_{\mathbb{BT}}$), 45]

49. $\neg\mathbf{in}(\boldsymbol{u_{\mathsf{bt}}}, \boldsymbol{\sigma_l}) \vee \sigma_k = \sigma_n \vee \mathsf{in}(\sigma_k, \sigma_l) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee$
$\vee\, \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A2}_{\mathbb{BT}}$), 46]

50. $\neg\mathbf{in}(\boldsymbol{u_{\mathsf{bt}}}, \boldsymbol{\sigma_r}) \vee \sigma_k = \sigma_n \vee \mathsf{in}(\sigma_k, \sigma_l) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee$
$\vee\, \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A2}_{\mathbb{BT}}$), 47]

51. $\boldsymbol{u_{\mathsf{bt}} \neq \sigma_n} \vee u_{\mathsf{bt}} < \sigma_k \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A2}_{\mathbb{BT}}$), 44]

52. $\neg\mathbf{in}(\boldsymbol{u_{\mathsf{bt}}}, \boldsymbol{\sigma_l}) \vee u_{\mathsf{bt}} < \sigma_k \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A2}_{\mathbb{BT}}$), 44]

53. $\neg\mathbf{in}(\boldsymbol{u_{\mathsf{bt}}}, \boldsymbol{\sigma_r}) \vee u_{\mathsf{bt}} < \sigma_k \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, u_{\mathsf{bt}}, \sigma_x))$ [BR ($\mathrm{A2}_{\mathbb{BT}}$), 44]

54. $\sigma_k = \sigma_n \vee \mathsf{in}(\sigma_k, \sigma_l) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \boldsymbol{\sigma_n}, \sigma_x))$ [ER 48]

55. $\sigma_n < \sigma_k \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \boldsymbol{\sigma_n}, \sigma_x))$ [ER 51]

56. $\sigma_n \neq \sigma_k \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR ($\mathrm{L12}_{\mathbb{N}}$), 55]

57. $\mathsf{in}(\sigma_k, \sigma_l) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR 54, 56]

58. $\neg(\sigma_w < \sigma_k) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR 40, 57]

59. $\neg(\sigma_w < \sigma_n) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR ($\mathrm{A9}_{\mathbb{N}}$), 55, 58]

60. $\neg(\sigma_w < \sigma_n) \vee \neg(\sigma'_w < \sigma_k) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR 42, 59]

61. $\boldsymbol{\neg(\sigma_w < \sigma_n)} \vee \boldsymbol{\neg(\sigma'_w < \sigma_n)} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_n, \sigma_x))$ [BR ($\mathrm{A9}_{\mathbb{N}}$), 55, 60]

62. $\sigma_k = \sigma_n \vee \mathsf{in}(\sigma_k, \sigma_l) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \boldsymbol{\sigma_w}, \sigma_x))$ [BR 39, 49, ER]

63. $\sigma_w < \sigma_k \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \boldsymbol{\sigma_w}, \sigma_x))$ [BR 39, 52, ER]

64. $\sigma_w < \sigma_n \vee \mathsf{in}(\sigma_k, \sigma_l) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [Sup 62, 63]

65. $\sigma_w < \sigma_n \vee \neg(\sigma_w < \sigma_k) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [BR 40, 64]

66. $\sigma_w < \sigma_n \vee \neg(\sigma_w < \sigma_k) \vee \neg(\sigma'_w < \sigma_k) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [BR 42, 65]

67. $\sigma_w < \sigma'_w \vee \neg(\sigma_w < \sigma'_w)$ [Taut.]

68. $\sigma_w < \sigma_n \vee \neg(\sigma_w < \sigma_k) \vee \sigma_w < \sigma'_w \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [BR ($\mathrm{L13}_{\mathbb{N}}$), 66, 67]

69. $\boldsymbol{\sigma_w < \sigma_n} \vee \boldsymbol{\sigma_w < \sigma'_w} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [BR 63, 68]

70. $\boldsymbol{\sigma'_w < \sigma_n} \vee \boldsymbol{\sigma_w < \sigma'_w} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [BR ($\mathrm{L13}_{\mathbb{N}}$), 69]

71. $\sigma_k = \sigma_n \vee \mathsf{in}(\sigma_k, \sigma_l) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \boldsymbol{\sigma'_w}, \sigma_x))$ [BR 41, 50, ER]

72. $\sigma'_w < \sigma_k \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \boldsymbol{\sigma'_w}, \sigma_x))$ [BR 41, 53, ER]

73. $\sigma'_w < \sigma_n \vee \mathsf{in}(\sigma_k, \sigma_l) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma'_w, \sigma_x))$ [Sup 71, 72]

74. $\sigma'_w < \sigma_n \vee \neg(\sigma_w < \sigma_k) \vee \mathsf{in}(\sigma_k, \sigma_r) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma'_w, \sigma_x))$ [BR 40, 73]

75. $\sigma'_w < \sigma_n \vee \neg(\sigma_w < \sigma_k) \vee \neg(\sigma'_w < \sigma_k) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma'_w, \sigma_x))$ [BR 42, 74]

76. $\sigma'_w < \sigma_n \vee \neg(\sigma'_w < \sigma_k) \vee \neg(\sigma_w < \sigma'_w) \vee$
$\vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma_w, \sigma_x))$ [BR ($\mathrm{L13}_{\mathbb{N}}$), 67, 75]

77. $\boldsymbol{\sigma'_w < \sigma_n} \vee \boldsymbol{\neg(\sigma_w < \sigma'_w)} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma'_w, \sigma_x))$ [BR 72, 76]

78. $\boldsymbol{\sigma_w < \sigma_n} \vee \boldsymbol{\neg(\sigma_w < \sigma'_w)} \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \sigma'_w, \sigma_x))$ [BR ($\mathrm{L13}_{\mathbb{N}}$), 77]

79. $\neg(\sigma'_w < \sigma_n) \vee \sigma_w < \sigma'_w \vee$
$\vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \textbf{if } \boldsymbol{\sigma_w < \sigma_n} \textbf{ then } \boldsymbol{\sigma_n} \textbf{ else } \boldsymbol{\sigma_w}, \sigma_x))$ [BR″ 61, 69]

80. $\sigma_w < \sigma'_w \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \textbf{if } \boldsymbol{\sigma'_w < \sigma_n} \textbf{ then}$
$\textbf{if } \boldsymbol{\sigma_w < \sigma_n} \textbf{ then } \boldsymbol{\sigma_n} \textbf{ else } \boldsymbol{\sigma_w} \textbf{ else } \boldsymbol{\sigma_w}, \sigma_x))$ [BR″ 70, 79]

81. $\neg(\sigma_w < \sigma_n) \vee \neg(\sigma_w < \sigma'_w) \vee$
$\vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \textbf{if } \boldsymbol{\sigma'_w < \sigma_n} \textbf{ then } \boldsymbol{\sigma_n} \textbf{ else } \boldsymbol{\sigma'_w}, \sigma_x))$ [BR″ 61, 77]

82. $\neg(\sigma_w < \sigma'_w) \vee \mathsf{ans}(\mathsf{rec}(\sigma_a, \textbf{if } \boldsymbol{\sigma_w < \sigma_n} \textbf{ then}$
$\textbf{if } \boldsymbol{\sigma'_w < \sigma_n} \textbf{ then } \boldsymbol{\sigma_n} \textbf{ else } \boldsymbol{\sigma'_w} \textbf{ else } \boldsymbol{\sigma'_w}, \sigma_x))$ [BR″ 78, 81]

83. $\mathsf{ans}(\mathsf{rec}(\sigma_a, \textbf{if } \boldsymbol{\sigma_w < \sigma'_w} \textbf{ then if } \boldsymbol{\sigma_w < \sigma_n} \textbf{ then}$
$\textbf{if } \boldsymbol{\sigma'_w < \sigma_n} \textbf{ then } \boldsymbol{\sigma_n} \textbf{ else } \boldsymbol{\sigma'_w} \textbf{ else } \boldsymbol{\sigma'_w} \textbf{ else if } \boldsymbol{\sigma'_w < \sigma_n} \textbf{ then}$
$\textbf{if } \boldsymbol{\sigma_w < \sigma_n} \textbf{ then } \boldsymbol{\sigma_n} \textbf{ else } \boldsymbol{\sigma_w} \textbf{ else } \boldsymbol{\sigma_w}, \sigma_x))$ [BR″ 80, 82]

84. $\square$ [answer literal removal 83]

The program we end up with is

$$f(x),$$

where

$$\begin{aligned}
f(\mathsf{leaf}(a)) &= a \\
f(\mathsf{bt}(l,n,r)) &= \mathsf{if}\ f(l) < f(r)\ \mathsf{then} \\
&\qquad \mathsf{if}\ f(l) < n\ \mathsf{then} \\
&\qquad\quad \mathsf{if}\ f(r) < n\ \mathsf{then}\ n\ \mathsf{else}\ f(r) \\
&\qquad \mathsf{else}\ f(r) \\
&\quad \mathsf{else if}\ f(r) < n\ \mathsf{then} \\
&\qquad\quad \mathsf{if}\ f(l) < n\ \mathsf{then}\ n\ \mathsf{else}\ f(l) \\
&\quad \mathsf{else}\ f(l).
\end{aligned}$$

Specification	Program	Synthesized definitions	VAMPIRE
Distributivity of multiplication 5.2: $\forall x_1, x_2, x_3 \in \mathbb{N}.\exists y \in \mathbb{N}.$ $(x_1 \cdot x_2 + x_1 \cdot x_3 = x_1 \cdot y)$	$f(x_3)$	$f(0) = x_2$ $f(\mathsf{s}(n)) = \mathsf{s}(f(n))$	✓
Subtraction with condition 7.1.1: $\forall x_1, x_2 \in \mathbb{N}.\exists y \in \mathbb{N}.$ $(x_2 < x_1 \rightarrow x_2 + y = x_1)$	$f(x_2)$	$f(0) = x_1$ $f(\mathsf{s}(n)) = \mathsf{p}(f(n))$	✓
Floored square root 7.1.2: $\forall x \in \mathbb{N}.\exists y \in \mathbb{N}.$ $(y \cdot y \leq x \wedge x < \mathsf{s}(y) \cdot \mathsf{s}(y))$	$f(x)$	$f(0) = 0$ $f(\mathsf{s}(n)) = \mathsf{if}\ \mathsf{s}(n) = \mathsf{s}(f(n)) \cdot \mathsf{s}(f(n))$ $\mathsf{then}\ \mathsf{s}(f(n))\ \mathsf{else}\ f(n)$	✗
Floored division 7.1.3: $\forall x_1, x_2 \in \mathbb{N}.\exists y \in \mathbb{N}.(x_2 \neq 0 \rightarrow$ $(y \cdot x_2 \leq x_1 \wedge x_1 < \mathsf{s}(y) \cdot x_2))$	$f(x_1)$	$f(0) = 0$ $f(\mathsf{s}(n)) = \mathsf{if}\ \mathsf{s}(n) = \mathsf{s}(f(n)) \cdot x_2$ $\mathsf{then}\ \mathsf{s}(f(n))\ \mathsf{else}\ f(n)$	✗
Length of 2 concatenated lists 7.2.1: $\forall x_1, x_2 \in \mathbb{L}.\exists y \in \mathbb{N}.$ $y = \mathsf{len}(x_1 +\!\!+ x_2)$	$f(x_1)$	$f(\mathsf{nil}) = \mathsf{len}(x_2)$ $f(\mathsf{cons}(n,l)) = \mathsf{s}(f(l))$	✓
Last element of a list 7.2.2: $\forall x \in \mathbb{L}.\exists y \in \mathbb{N}.(x \neq \mathsf{nil} \rightarrow$ $\exists z \in \mathbb{L}.x = z +\!\!+ \mathsf{cons}(y, \mathsf{nil}))$	$f(x)$	$f(\mathsf{cons}(n,\mathsf{nil})) = n$ $l \neq \mathsf{nil} \rightarrow f(\mathsf{cons}(n,l)) = f(l)$	✓
Prefix of a list given its suffix 7.2.3: $\forall x_1, x_2 \in \mathbb{L}.\exists y \in \mathbb{L}.$ $(\mathsf{suff}(x_2, x_1) \rightarrow x_1 = y +\!\!+ x_2)$	$f(x_2)$	$f(\mathsf{nil}) = x_1$ $f(\mathsf{cons}(n,l)) = g(f(l))$ $g(\mathsf{cons}(n,\mathsf{nil})) = \mathsf{nil}$ $l \neq \mathsf{nil} \rightarrow g(\mathsf{cons}(n,l)) = \mathsf{cons}(n, g(l))$	✗
Maximum element of a list 7.2.4: $\forall x \in \mathbb{L}.\exists y \in \mathbb{N}.(x \neq \mathsf{nil} \rightarrow$ $(\mathsf{in}(y,x) \wedge \forall k \in \mathbb{N}.(\mathsf{in}(k,x) \rightarrow k \leq y))$	$f(x)$	$f(\mathsf{cons}(n,\mathsf{nil})) = n$ $l \neq \mathsf{nil} \rightarrow f(\mathsf{cons}(n,l)) = \mathsf{if}\ f(l) < n$ $\mathsf{then}\ n\ \mathsf{else}\ f(l)$	✗
Maximum element of a tree 7.3.1: $\forall x \in \mathbb{BT}.\exists y \in \mathbb{N}.$ $(\mathsf{in}(y,x) \wedge \forall k \in \mathbb{N}.(\mathsf{in}(k,x) \rightarrow k \leq y)$	$f(x)$	$f(\mathsf{leaf}(n)) = n$ $f(\mathsf{bt}(l,n,r)) =$ $\mathsf{if}\ f(l) < f(r)\ \mathsf{then}$ $\mathsf{if}\ f(l) < n\ \mathsf{then}$ $\mathsf{if}\ f(r) < n\ \mathsf{then}\ n\ \mathsf{else}\ f(r)$ $\mathsf{else}\ f(r)$ $\mathsf{else if}\ f(r) < n\ \mathsf{then}$ $\mathsf{if}\ f(l) < n\ \mathsf{then}\ n\ \mathsf{else}\ f(l)$ $\mathsf{else}\ f(l)$	✗

Table 7.1: Summary of all introduced 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 the introduced framework synthesizes all these examples, the implementation in VAMPIRE only synthesizes those marked with "✓". Note that for "Length of 2 concatenated lists" we consider $+\!\!+$ to be uncomputable.

$$\forall x \in \mathbb{N}.\ \mathsf{s}(x) \neq 0 \tag{A1$_\mathbb{N}$}$$
$$\forall x, y \in \mathbb{N}.\ x = y \to \mathsf{s}(x) = \mathsf{s}(y) \tag{A2$_\mathbb{N}$}$$
$$\forall x \in \mathbb{N}.\ x + 0 = x \tag{A3$_\mathbb{N}$}$$
$$\forall x, k \in \mathbb{N}.\ x + \mathsf{s}(k) = \mathsf{s}(x + k) \tag{A4$_\mathbb{N}$}$$
$$\forall x \in \mathbb{N}.\ x \cdot 0 = 0 \tag{A5$_\mathbb{N}$}$$
$$\forall x, n \in \mathbb{N}.\ x \cdot \mathsf{s}(n) = x \cdot n + x \tag{A6$_\mathbb{N}$}$$
$$\forall x \in \mathbb{N}.\ \neg(x < 0) \tag{A7$_\mathbb{N}$}$$
$$\forall x, y \in \mathbb{N}.\ x < \mathsf{s}(y) \leftrightarrow x < y \vee x = y \tag{A8$_\mathbb{N}$}$$
$$\forall x, y, z \in \mathbb{N}.\ (x < y \wedge y < z) \to x < z \tag{A9$_\mathbb{N}$}$$

$$\forall x \in \mathbb{L}.\ x = x{+\!\!+}\mathsf{nil} \tag{A1$_\mathbb{L}$}$$
$$\forall x, l \in \mathbb{L}.\ \forall n \in \mathbb{N}.\ \mathsf{cons}(n, l){+\!\!+}x = \mathsf{cons}(n, l{+\!\!+}x) \tag{A2$_\mathbb{L}$}$$
$$\mathsf{len}(\mathsf{nil}) = 0 \tag{A3$_\mathbb{L}$}$$
$$\forall l \in \mathbb{L}.\ \forall n \in \mathbb{N}.\ \mathsf{len}(\mathsf{cons}(n, l)) = \mathsf{s}(\mathsf{len}(l)) \tag{A4$_\mathbb{L}$}$$
$$\forall l \in \mathbb{L}.\ \mathsf{suff}(\mathsf{nil}, l) \tag{A5$_\mathbb{L}$}$$
$$\forall l \in \mathbb{L}.\ \forall n \in \mathbb{N}.\ \neg(\mathsf{suff}(\mathsf{cons}(n, l), \mathsf{nil})) \tag{A6$_\mathbb{L}$}$$
$$\forall l, l' \in \mathbb{L}.\ \forall n \in \mathbb{N}.\ \mathsf{suff}(l', l) \to \mathsf{suff}(l', \mathsf{cons}(n, l)) \tag{A7$_\mathbb{L}$}$$
$$\forall l, l' \in \mathbb{L}.\ \forall n' \in \mathbb{N}.\ \mathsf{suff}(\mathsf{cons}(n', l'), l) \to \mathsf{suff}(l', l) \tag{A8$_\mathbb{L}$}$$
$$\forall l, l' \in \mathbb{L}.\ \forall n, n' \in \mathbb{N}.\ \mathsf{cons}(n, l) = \mathsf{cons}(n', l') \to (n = n' \wedge l = l') \tag{A9$_\mathbb{L}$}$$
$$\forall n \in \mathbb{N}.\ \neg\mathsf{in}_\mathbb{L}(n, \mathsf{nil}) \tag{A10$_\mathbb{L}$}$$
$$\forall l \in \mathbb{L}.\ \forall n, k \in \mathbb{N}.\ \mathsf{in}_\mathbb{L}(n, \mathsf{cons}(k, l)) \leftrightarrow (\mathsf{in}_\mathbb{L}(n, l) \vee n = k) \tag{A11$_\mathbb{L}$}$$

$$\forall n, k \in \mathbb{N}.\ \mathsf{in}_{\mathbb{BT}}(n, \mathsf{leaf}(k)) \leftrightarrow n = k \tag{A1$_{\mathbb{BT}}$}$$
$$\forall l, r \in \mathbb{BT}.\ \forall n, k \in \mathbb{N}.\ \mathsf{in}_{\mathbb{BT}}(n, \mathsf{bt}(l, k, r)) \leftrightarrow (\mathsf{in}_{\mathbb{BT}}(n, l) \vee \mathsf{in}_{\mathbb{BT}}(n, r) \vee n = k) \tag{A2$_{\mathbb{BT}}$}$$

Figure 7.2: List of all axioms defining function and predicate symbols and/or are used for the derivations.

$$\forall x \in \mathbb{N}.\ x = 0 + x \tag{L1$_\mathbb{N}$}$$
$$\forall x, n \in \mathbb{N}.\ \mathsf{s}(x) + n = x + \mathsf{s}(n) \tag{L2$_\mathbb{N}$}$$
$$\forall x \in \mathbb{N}.\ 0 \cdot x = 0 \tag{L3$_\mathbb{N}$}$$
$$\forall x, y \in \mathbb{N}.\ \mathsf{s}(x) \cdot y = x \cdot y + y \tag{L4$_\mathbb{N}$}$$
$$\forall x, y \in \mathbb{N}.\ \mathsf{s}(x) < y \to x < y \tag{L5$_\mathbb{N}$}$$
$$\forall x, y, z \in \mathbb{N}.\ (y + x = z \wedge y < z) \to x \neq 0 \tag{L6$_\mathbb{N}$}$$
$$\forall x, y \in \mathbb{N}.\ x < y \to x \neq y \tag{L7$_\mathbb{N}$}$$
$$\forall x, y \in \mathbb{N}.\ x < y \to (\mathsf{s}(x) < y \vee \mathsf{s}(x) = y) \tag{L8$_\mathbb{N}$}$$
$$\forall x, y \in \mathbb{N}.\ y \leq x \to y \leq \mathsf{s}(x) \tag{L9$_\mathbb{N}$}$$
$$\forall x, y \in \mathbb{N}.\ y \neq 0 \to x < x + y \tag{L10$_\mathbb{N}$}$$
$$\forall x, y \in \mathbb{N}.\ x < y \to x \leq y \tag{L11$_\mathbb{N}$}$$
$$\forall x, y \in \mathbb{N}.\ x < y \to x \neq y \tag{L12$_\mathbb{N}$}$$
$$\forall x, y, z \in \mathbb{N}.\ ((x \leq y \vee x \leq z) \wedge y \leq z) \to x \leq z \tag{L13$_\mathbb{N}$}$$
$$\forall n, n', m \in \mathbb{N}.\ n = n' \to m + n = m + n' \tag{L14$_\mathbb{N}$}$$

$$\forall x \in \mathbb{L}.\ \mathsf{nil}{+\!\!+}x = x \tag{L1$_\mathbb{L}$}$$
$$\forall l, l' \in \mathbb{L}.\ \forall n \in \mathbb{N}.\ l' \neq \mathsf{cons}(n, l) \to (\mathsf{suff}(l', \mathsf{cons}(n, l)) \to \mathsf{suff}(l', l)) \tag{L2$_\mathbb{L}$}$$
$$\forall x, y, z \in \mathbb{L}.\ \forall n \in \mathbb{N}.\ (x = y{+\!\!+}z \wedge \mathsf{suff}(\mathsf{cons}(n, z), x)) \to \mathsf{suff}(\mathsf{cons}(n, \mathsf{nil}), y) \tag{L3$_\mathbb{L}$}$$
$$\forall x, y \in \mathbb{L}.\ n \in \mathbb{N}.\ \mathsf{suff}(\mathsf{cons}(n, x), y) \to x \neq y \tag{L4$_\mathbb{L}$}$$
$$\forall x, y, z \in \mathbb{L}.\ (x = y{+\!\!+}z \wedge x \neq z) \to y \neq \mathsf{nil} \tag{L5$_\mathbb{L}$}$$
$$\forall x, y, z \in \mathbb{L}.\ x = y \to x{+\!\!+}z = y{+\!\!+}z \tag{L6$_\mathbb{L}$}$$

Figure 7.3: List of all lemmas used for the derivations.

8 Related Work

In this chapter, several frameworks for synthesizing non-recursive and recursive programs are compared.

Deductive Synthesis with Answer Literals. The approach introduced in this thesis builds upon non-recursive synthesis [HKNV23] using answer literals [Gre69] and extends this framework to synthesize recursive functions in the program with the use of special induction axioms, dubbed magic axioms [HAH+24]. This framework extends upon a theorem prover with saturation. This requires a fully formalized specification in first-order logic and is fully automated.

A similar approach was developed by Manna and Waldinger [MW80] that relies on the one-to-one translation of induction axioms and recursive functions.

In [LWC74] results stemming from Waldinger and Lee are further developed to increase efficiency. The correctness proof of the introduced algorithm is given.

A form of completeness is proven for the A-resolution calculus in [Tam95] that was also used for adapting the superposition calculus rules in [HKNV23].

Unlike [MW80, LWC74, Tam95], we use answer literals in saturation-based proving with induction, allowing us to synthesize recursive functions from superposition proofs.

Component-Based Synthesis. Component-based synthesis constructs functions using a set of libraries and differs in this regard to deductive synthesis. The usage of APIs from libraries has the effect that the correctness of the constructed program is no longer a given.

In [GJTV11] synthesis of loop-free programs is explored using components from a given library. This is done using a constraint-based approach; the algorithm introduced uses SMT solvers.

Graphical specifications are used to synthesize programs from a subroutine library in [SWL+94]. Applications consist of constructing software for interplanetary missions. Graphical specification in this sense means, that the specification is not formally described but is drawn out by the user through a menu-driven graphical user interface, which has been very well received.

The work in [TGD15] is also based on SMT solvers and uses two distinct interpretations of symbols occurring in the program.

Unlike these works, our approach is not restricted to decidable SMT theories and does not require user guidance in the function specification.

Syntax-Guided Synthesis (SyGuS). The main idea of Syntax-guided synthesis is to decrease the possible search space of programs by allowing the user to, additionally to a logical specification, give a syntactic set of possible implementations, [ABD+15]. The

problem then is to find a solution for the specification that also fulfills the given constraints stated by the user.

In contrast to this line of research, we do not rely on user-given constraints or templates.

Sketching Technique. The sketching technique [SL09] allows users to partially state the intended meaning of the program aimed to be synthesized. The synthesis task is then to fill this sketch of a program to a full extent. This approach does not rely on logical specifications as an input.

ROSETTE, a framework that designs solver-aided languages is introduced in [TB13] and relies on the sketching technique.

In [TNS+21] one specifies the intended program behavior by input/output examples which also differs from our approach. This is called example-guided synthesis; a well-known example of this approach is FlashFill which uses programming by example in Microsoft Excel [Gul11].

Our approach is conceptually different as we synthesize code from its correctness proof, using superposition reasoning and without concrete instances of input/output examples. Rather, we use the logical specification of the code to be synthesized.

9 Conclusions

In this thesis, we introduced the basic resolution calculus for theorem proving in propositional logic (see Section 2.1) as well as first-order logic (see Section 2.2) in Chapter 2. We provided small examples to enhance comprehension of the newly established concepts. After that, we introduced the superposition calculus that is based on the resolution calculus of first-order logic and used for reasoning over theories with equality in Section 2.3. We have proven the soundness of the superposition calculus.

To further deepen the understanding of how the main inference rules of the superposition calculus act, we have shown a derivation of the distributivity example and explained it in detail in Chapter 3.

As a next step in Chapter 4 we explained the non-recursive synthesis approach from [HKNV23] in detail (see Section 4.1) and further demonstrated how the introduced framework solves a small example, finding the maximum of two natural numbers (see Section 4.2). We also pointed out the limitations of the framework regarding recursive synthesis in Section 4.3.

Based on the distributivity example, we explained thoroughly how the extended framework for synthesizing recursive programs over the natural numbers [HAH+24] works in Chapter 5. This entailed changing the prioritizing order of the quantifiers while transforming the specification into prenex normal form (PNF). The quantifiers are therefore moved to the front in such a way, that the existentially quantified output variable y depends on three arguments; the input variable x, the base case variable u_0 and the step case variable u_s. After skolemizing, this produced a skolem-function, denoted with rec, that depends on these three arguments. When the rec-term is resolved it appears inside the answer literal and is used to construct a recursive function when the derivation terminates.
We explained this derivation process in detail based on the distributivity example in Section 5.2.
Further, we showed the output for this specific example of the synthesis implementation of the first-order theorem prover Vampire. This was done to display the correspondence between manual proof derivations and automatic proofs.

We introduced the concepts of inductive structures, inductive proofs, and unique readability in Section 6.1 of Chapter 6. These concepts helped us establish the unique translation from rec terms to recursive functions, [GJ95].
Further, we introduced the notions of magic axioms to help synthesize recursive functions and explained the adapted rules for the recursive synthesis framework. We introduced these rules in order to substitute if − then − else−constructors into rec-terms.

We introduced inductive structures, namely natural lists and natural binary trees, and explained the structure of the magic axioms for these types of inductive structures.

In Chapter 7 we gave program derivations for interesting examples over different inductive structures and translated the output into recursive programs, see Table 7.1.

To showcase several applications of induction and how this translates into several recursive functions that are internally linked we presented the examples *subtraction with condition* and *prefix of a list*.

With the examples *rounded root* and *rounded division* we explored how the framework synthesizes a recursive function with given constraints as an input.

The examples *maximum of a list* and *maximum of a binary tree* show how fast the search space continues to grow when induction is applied on a formula containing multiple clauses and/or literals.

Finally, we compared different synthesis techniques in Chapter 8. In particular, deductive synthesis with the use of answer literals, component-based synthesis, syntax-guided synthesis, and the sketching technique, which also includes programming by example.

Bibliography

[ABD+15] Rajeev Alur, Rastislav Bodik, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shamwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-Guided Synthesis*, volume 40 of *Nato Science for Peace and Security Series D-Information and Communication Security*, pages 1–25. 2015.

[Cok13] D. R. Cok. The smt-libv2 language and tools: A tutorial. 2013.

[DV01] Anatoli Degtyarev and Andrei Voronkov. Chapter 10 - equality reasoning in sequent-based calculi. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 611–706. North-Holland, Amsterdam, 2001.

[GGH18] Martin Goldstern, Moritz Gschwandtner, and Stefan Hetzl. *Logik und Grundlagen*. Vienna, 2018.

[GJ95] Martin Goldstern and Haim Judah. *The Incompleteness Phenomenon. A New Course in Mathematical Logic*. A.K.Peters, Boston, 1995.

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *SIGPLAN Not.*, 46(6):62–73, jun 2011.

[GPS17] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[Gre69] Cordell Green. Theorem-Proving by Resolution as a Basis for Question-Answering Systems. *Machine Intelligence*, 4:183–205, 1969.

[Gul11] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, 46(1):317–330, jan 2011.

[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.

[HKNV23] Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov. Program Synthesis in Saturation. EasyChair Preprint no. 10223, EasyChair, 2023.

[HKRV22] Márton Hajdu, Laura Kovács, Michael Rawson, and Andrei Voronkov. The Vampire Approach to Induction. In *Practical Aspects of Automated Reasoning*, 2022.

[KV13] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, pages 1–35, 2013.

[LWC74] R. C. T. Lee, R. J. Waldinger, and C. L. Chang. An Improved Program-Synthesizing Algorithm and Its Correctness. *Commun. ACM*, (4):211–217, 1974.

[MW80] Zohar Manna and Richard Waldinger. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[NR01] Robert Nieuwenhuis and Albert Rubio. Chapter 7 - paramodulation-based theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 371–443. North-Holland, Amsterdam, 2001.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, jan 1965.

[SL09] Armando Solar-Lezama. The sketching approach to program synthesis. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pages 4–13, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[SWL+94] Mark Stickel, Richard Waldinger, Michael Lowry, Thomas Pressburger, and Ian Underwood. Deductive composition of astronomical software from subroutine libraries. In Alan Bundy, editor, *Automated Deduction — CADE-12*, pages 341–355, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[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 *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery.

[TGD15] Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. Program synthesis using dual interpretation. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 482–497, Cham, 2015. Springer International Publishing.

[TNS+21] Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. Example-guided synthesis of relational queries. pages 1110–1125, 06 2021.