TU WIEN Informatics

Automating inductive reasoning with recursive functions

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Márton Hajdu, BSc.
Matrikelnummer 11849197

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ.Prof. Dr.techn. Laura Kovács, MSc

Wien, 4. November 2020

Márton Hajdu	Laura Kovács

TU WIEN Informatics

Automating inductive reasoning with recursive functions

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Márton Hajdu, BSc.
Registration Number 11849197

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr.techn. Laura Kovács, MSc

Vienna, 4th November, 2020

Márton Hajdu	Laura Kovács

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

Erklärung zur Verfassung der Arbeit

Márton Hajdu, BSc.

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

Wien, 4. November 2020

Márton Hajdu

Acknowledgements

I would like to thank my supervisor Laura Kovács for her enthusiasm and ideas and for guiding me through the process of writing this thesis. For all the ideas and help with theoretical and implementation aspects that I received, I am very grateful to Petra Hozzová, Andrei Voronkov, Jakob Rath, Giles Reger, Martin Suda and Bernhard Gleiss. I also thank Johannes Schoisswohl for generating most of the benchmarks that I used. Finally, I am thankful for my family and friends who supported and motivated me during this time.

The work in this thesis has been partially supported by the ERC starting grant SYMCAR 639280 and the ERC proof of concept grant SYMELS 842066.

Abstract

Formally ensuring software reliability is challenging due to the growing complexity of software in terms of its size, used unbounded and inductive data types and supported workflow. As manual code analysis is not a viable solution, automated reasoning approaches are needed for proving software correct. There are several prominent methods addressing this challenge by (i) translating parts of a system to a logic-based formalism and (ii) using automated theorem proving to validate certain safety and liveness program properties.

In this context a vital ingredient in automated reasoning is inductive reasoning, which can/must be used whenever repetition is present in a software component, e.g. loops or recursive data structures in computer programs or feedback loops in electric circuits. The two main branches in the literature are explicit and implicit induction. In this thesis focus on the former approach which is based broadly on the Noetherian well-founded induction principle. There is a plethora of research already done in this area with several automated theorem provers as well as proof assistants are equipped with inductive reasoning techniques. Some of the earliest ones are Nqthm/ACL2, Inka and Oyster/CLAM. Recently, modern automated theorem provers, such as ZipperPosition, CVC4 and Vampire, started to incorporate inductive types and reasoning.

The aim of the proposed thesis is to advance the state-of-the art in inductive reasoning within first-order theorem proving, by using automating reasoning with recursive function definitions over inductive data types.

Contents

CHAPTER 1

Introduction

Nowadays, due to explosive amounts of free information, advances in computer technologies and an increasing need for automation in almost every field from manufacturing to research, the dependence of our society on software and hardware systems is indisputable. There is an ever-emerging need for robustness and reliability in large-scale systems where downtime and errors can put lives at risk or cause significant economical loss.

Complexity of most systems with reliability as a first priority is already at a level that makes checking software correctness or detecting software bugs out of reach for humans, giving rise to the automation of these tasks. Several prominent methods for doing this rigorously involve translating parts of a system to a logic-based formalism [Hoa69, CE82, HKV11] and using automated theorem proving to deduce certain safety and liveness properties or a lack thereof [DLL62, QS82].

In this context a vital ingredient is inductive reasoning [BM79], which can be used whenever repetition is present in a component, e.g. loops or recursive data structures in computer programs or feedback loops in electric circuits. The two main branches in the literature are explicit and implicit induction. We focus on the former which is based broadly on the Noetherian well-founded induction principle. There is a plethora of research already done in this area [Bun01, MW13] with several automated theorem provers as well as proof assistants are equipped with inductive reasoning techniques. Some of the earliest ones are Nqthm/ACL2 [BM79, BM88], Inka [BHHW86] and Oyster/CLAM [BvHHS90]. Recently, modern automated theorem provers using e.g. superposition started to incorporate inductive types as well [Cru17, RV19]. These include ZipperPosition [Cru17], CVC4 [BCD+11] and Vampire [KV13] to list a few. *The aim of the proposed thesis is to advance the state-of-the-art in inductive reasoning within first-order theorem proving.*

1.1 Problem statement

A functional program can contain various function definitions some of which may call themselves, these are called recursive functions. As a motivating example, consider the following function:

$$\begin{aligned}&\textbf{fun } \texttt{add\_even}(x \in \mathbb{N}, y \in \mathbb{N}) \to \mathbb{N} := \\ &\qquad \textbf{if } (x \texttt{ mod } 2) = (y \texttt{ mod } 2) \textbf{ then } x + y \\ &\qquad\qquad \textbf{else } x + y + 1\end{aligned}$$

This function `add_even` returns the first even natural number that is greater than or equal to the sum of its arguments. Now we may want to prove functional correctness of this function by stating that for any two natural numbers x and y, the function `add_even` indeed returns an even number and this number is at least the sum of the two:

$$\forall x, y.(\texttt{add\_even}(x, y) \texttt{ mod } 2) = 0 \wedge x + y \leq \texttt{add\_even}(x, y)$$

This can be proven by induction if we define natural numbers as an inductive type `nat` consisting of the base constructor `0` and recursive constructor `s`. Then, usually the recursive function definitions for $+$ (`add`) and `mod` 2 (`even`) are given by the following axioms:

$$\begin{aligned}\forall y.\texttt{add}(\texttt{0}, y) &= y \\ \forall x, y.\texttt{add}(\texttt{s}(x), y) &= \texttt{s}(\texttt{add}(x, y))\end{aligned}$$

$$\begin{aligned}\texttt{even}(\texttt{0}) \\ \neg\texttt{even}(\texttt{s}(\texttt{0})) \\ \forall y.\texttt{even}(\texttt{s}(\texttt{s}(y))) &\leftrightarrow \texttt{even}(y)\end{aligned}$$

From now on, we may use these function symbols and their recursive definitions interchangably. Representing recursive functions as ordinary quantified (and maybe guarded) equations, however, is not effective in two regards.

First, advanced techniques for selecting a usable induction scheme for a particular subgoal often need special treatment of the inductive function definition (example given later).

Second, a function definition header in a goal to be proven is typically expanded into its function definition body during the refutation process. As saturation-based first-order provers perform rewriting according to a specific term ordering [BG94], this may only be possible if we select always the "right" inductive definitions for a function header, otherwise there could be too many options to choose from leading to a blow-up in the search space. In the example above, there could be other input axioms e.g.:

$$\forall x.\texttt{add}(\texttt{s}(\texttt{0}), x) = \texttt{s}(x)$$

or

$$\forall x, y.\texttt{even}(\texttt{add}(x, y)) \leftrightarrow \texttt{even}(\texttt{add}(\texttt{s}(x), \texttt{s}(y)))$$

Although these provide additional information about the defined functions which the prover could use right away, we should not mix them with the definitional axioms of the functions. Coming up with heuristics that select from an input set of axioms the most general subset of definitional axioms which subsume the rest is not straightforward. Now consider the following conjecture:

$$\forall x, y.y + (x + x) = (x + y) + x$$

To prove this conjecture, we need to create an appropriate induction scheme which will allow us to prove all base cases and simplify the inductive step cases in a way that the corresponding induction hypotheses apply to the inductive step consequent. This involves selecting induction terms, choosing occurrences for these and creating base and step cases. In the example above, we can select x, y or both and various combinations of occurrences for both variables as induction terms. For instance, one possible induction scheme is the following, with selected variable y, inducting on all its occurrences and using base case $\mathtt{0}$ and step case $z \to \mathtt{s}(z)$:

$$\forall x. \left(\begin{array}{c} \mathtt{0} + (x + x) = (x + \mathtt{0}) + x \quad \wedge \\ \forall z. \left(\begin{array}{c} z + (x + x) = (x + z) + x \to \\ \mathtt{s}(z) + (x + x) = (x + \mathtt{s}(z)) + x \end{array} \right) \end{array} \right) \to \forall y.y + (x + x) = (x + y) + x$$

The problem of creating an appropriate induction scheme is equivalent to instantiating the Noetherian induction scheme with all possible orders and then checking each order for well-foundedness, which is known to be undecidable [Tur37, G̈31]; so we can only apply either all possible induction schemes exhaustively or use heuristics to find "good" ones with a relatively high success rate. There already exist heuristics which address this and perform relatively well for certain inductive problems – see for example recursion analysis, which dates back to the earliest version of ACL2 [BM88] or the heuristics of Aubin [Aub77] and Walther [Wal92, Wal93, Wal94]. Further, the metaheuristics of rippling generalize these approaches to any mathematical proof [BBHI05]. These metaheuristics are implemented in e.g. the proof planner of Isabelle called IsaPlanner [DF03] which provides the basis for proof assistants and automated theorem provers such as Hipster and HipSpec [LVJ15, CJRS13].

Problem statement. The thesis focuses on further extending and improving automated first-order theorem proving with inductive reasoning as follows:

G1. Find and compare techniques from the literature to automatically generate and combine induction formulas for inductive goals. Apply these techniques in the context of saturation-based first-order theorem proving.

G2. Modify the current superposition calculus with inference rules that allow expansion of inductive function headers into their definitions, while trying to maintain refutational completeness.

G3. Extend the standard input syntax of first-order provers – and in particular of the Vampire prover – such that the user can define recursive functions as special functions. Explore ideas to preprocess these functions to help the refutation process of saturation-based first-order theorem proving.

G4. Design new techniques to automatically discover relevant function definitional information from axioms.

Methodology

We focus our approach on developing methods for saturation-based theorem proving [KV13, RV01]. For supporting our work, we will use the state-of-the-art superposition theorem prover Vampire.

In saturation-based theorem proving, we start from the input axioms for a particular theory together with the negated goal, convert it to clausal normal form and try to saturate the search space, i.e. derive all possible consequences of the input based on a set of inference rules – in our case, the *superposition calculus* [RV01]. If we can derive the empty clause, the original goal is proven. An *inductive type* contains at least one *base constructor* (a 0-ary function or constant) and *recursive constructors* (n-ary function ($n > 0$) with at least one argument that has the same inductive type). In the previous examples, the set of natural numbers contains `0` as base and `s`(x) as recursive constructor.

A *recursive function* has at least one inductive type as an argument on which it recurses, that is, it calls itself with a subterm of that argument. It also has base cases where the recursion stops. This distinction between recursion and a base case is usually dependent on a condition or on case distinction in the input inductive arguments based on the inductive type's constructors. An *inductive formula*, and in particular an *inductive goal* is any formula containing inductive variables or terms and recursive functions.

Addressing our G3, *we extend the current input format of first-order reasoners* to allow user-defined recursive function definitions. From the available input formats, SMT-LIB [BFT16] supports recursive function definitions with the `define-fun-rec` and `define-funs-rec` keywords. Moreover, as recursive function definitions require case

distinctions between the different constructors of a recursive type and to ease the problem specification for users and increase readability, we also implement the `match` framework of SMT-LIB.

By aiming to solve G2, we note that for the superposition calculus to be complete, a *well-founded term ordering* is used so that there can be no infinite derivations. On the other hand, in order to prove conjectures about arbitrary recursive functions, we must be able to expand function headers into their definitions. This, however, may be in conflict with the ordering. E.g. instead of the above definition for the recursive case of `add`, we can create the following with the additional `p` predecessor destructor function:

$$\forall x, y.x \neq \mathtt{0} \rightarrow \mathtt{add}(x, y) = \mathtt{s}(\mathtt{add}(\mathtt{p}(x), y))$$

This function axiom has a right hand side in its consequent equality containing more terms than its header, so the superposition calculus may be unable to expand any such header occurrences into the body, which is often needed to proceed with a proof. In this thesis, *we extend the superposition calculus with exceptions for recursive function definitions*, such that the necessary function header expansions can be done to complete a proof.

By targeting G1 and G4, we emphasize that for a particular recursive function, we can create a set containing the function header, recursive calls and conditions tuples based on the branching and case distinctions in the function definition. For the `add` function, the set is the following:

$$\left\{ \begin{array}{ccc} (\mathtt{add}(\mathtt{0}, y), & \emptyset, & \emptyset) \\ (\mathtt{add}(\mathtt{s}(x), y), & \{\mathtt{add}(x, y)\}, & \emptyset) \end{array} \right\}$$

This set is called the *induction template* of the function `add`. The argument positions in the function term of a first tuple element that are non-variables and differ from the same argument position in some of the function terms in the corresponding second tuple element are called the *active positions* of the function. A term in an inductive goal that is a recursive function occurrence, which is itself in an active position of another recursive function occurrence or at a top-level position is a candidate for such analysis and is called an *active occurrence* of the term [Aub77, Cru17]. For example, in the conjecture from earlier $\forall x, y.y + (x + x) = (x + y) + x$, both $y + (x + x)$ and $(x + y) + x$ are top-level terms, then from the first term y is in an active position, from the second $(x + y)$.

By analyzing a particular inductive goal based on the templates for its function occurrences, we can instantiate one or more *induction schemes* suitable for proving the goal. As described earlier we can select any number of variable (or term) occurrences based on some heuristics. Then, an induction scheme can be stored in a compact form similar to that of the induction template, where we associate the selected variables/terms with the appropriate substitutions for each base/recursive case of the induction template. For the running example, one such scheme is:

$$\left\{ \begin{array}{ccc} (\{x \mapsto \mathtt{0}, y \mapsto \mathtt{0}\}, & \emptyset, & \emptyset) \\ (\{x \mapsto \mathtt{s}(z), y \mapsto \mathtt{s}(w)\}, & \{\{x \mapsto z, y \mapsto w\}\}, & \emptyset) \end{array} \right\}$$

which will then result in the following induction formula, where we only replace the active occurrences of x and y:

$$\forall x, y. \begin{pmatrix} \mathtt{0} + (x + x) = (\mathtt{0} + y) + x \quad (1) \quad \wedge \\ \forall z, w. \begin{pmatrix} w + (x + x) = (z + y) + x \rightarrow \\ \mathtt{s}(w) + (x + x) = (\mathtt{s}(z) + y) + x \end{pmatrix} (2) \end{pmatrix} \rightarrow \forall z, w. w + (x + x) = (z + y) + x$$

This induction formula contains a base case indicated by (1) and a recursive case (2) where the implicant is the *induction hypothesis*, whereas the implication conclusion is the *inductive step*.

Addressing G1, *we devise succinct representation and analysis of induction templates and induction schemes* and experiment with these. Heuristics must be also devised to deal with *the selection of induction terms and automatic generation of induction formulas*. These should be able to generalize over multiple variable occurrences or complex terms in order to solve more complex problems. Also, to this end, a diverse set of inductive problems must be prepared to conduct experiments upon. *We create heuristics to discover the smallest set of axioms defining functions and predicates, find sufficient conditions for their well-foundedness and well-definedness, moreover orient equalities for using them as function definitions*, targeting G4.

1.2 Relation to the State-of-the-Art

Automating inductive reasoning had several promising research areas in the recent decades including deciding when to apply induction and what induction formulas to use [BBHI05, RV19, CJRS13], restricting the fragment of logic [BM88, RV19, KP13, EP20], supporting the proof with auxiliary lemmas [CJRS13, Cru17] or circumventing the need for induction formulas and lemmata [BS10, BIS92]. Inductive reasoning automation dates back to the 1970s, when the first version of interactive theorem prover ACL2 [BM79, BM88] was created which later proved to be very useful in proving many theorems including correctness of microprocessors or Gödel's Incompleteness Theorem [G31̈]. Although proving in ACL2 is not fully automated, it incorporated the first version of the so-called "recursion analysis" technique, providing the user with a set of possible induction schemes for each subgoal.

Other interactive theorem provers (proof-assistants) such as Coq [Tea09], HipSpec/Hipster [CJRS13, LVJ15] or Isabelle [NPW02] also support induction with different levels of automation, such as rippling, which is a set of guidelines to measure the progress of induction and how far away is the prover from applying the induction hypotheses in the inductive step conclusions. Until recently however, the only automated provers that supported in general structural induction were either incomplete or inefficient in doing so. Some non-superposition based provers with inductive capabilities include CVC4 [BCD+11] – an SMT (Satisfiability Modulo Theories) solver [RK15] – or Zeno [SDE12] – which proves recursive Haskell program properties. While the former effectively deals with certain theories such as natural numbers but fails to extend inductive reasoning to

more complex data types such as lists or binary trees, the latter extends to arbitrary data types but fails to scale to problems greater in size.

Most state-of-the-art first order theorem provers with equality are using the superposition calculus. There have been several attempts recently at extending superposition with induction, see e.g. [KP13, EP20]. Some of the first superposition-based provers to support inductive datatypes and induction were ZipperPosition [Cru17] and Vampire [KV13]. ZipperPosition performs relatively well for a great set of input problems but is claimed to be incomplete due to its combination of recursive definitions as rewrite rules with superposition. In Vampire, several induction techniques are implemented for both structural and mathematical induction [RV19], however none of these use the structure of recursive definitional axioms to generate better induction formulas. Generalization over induction variables is now also supported in Vampire [HHK+20] which creates lots of induction formulas that could be discarded in light of the recursive function definitions structure. *These recent developments in Vampire for inductive reasoning will provide the basis of this thesis.*

1.3 Contributions

The contributions of this thesis are the following:

C1. Extract *case distinctions*, *argument orderings* and *recursive calls* from function definitions. When given a formula containing a particular recursive function symbol, *generate induction formulae* for this that match the function definition. Use argument orderings and other information from the formula to *strengthen induction hypotheses*. In case of multiple function symbols in a formula, *combine the extracted information* such that a minimum number of induction formulae are generated, each matching as many function symbols as possible, with the least amount of intersections. These are described in Chapter 3.

When applied in the context of saturation-based theorem proving, add a new inference rule called *multi-clause induction* to be able to induct on formulae more complex than a single literal as described in Chapter 4.

C2. Extend the superposition calculus with *function definition rewriting* with the least amount of disruption in refutational completeness as described in Chapter 4. That is, apply *demodulation* whenever a function definition orientation cioncides with the ordering used by the prover, otherwise use a modified *paramodulation* that expands the function header into its definition.

C3. Implement the keyword `define-fun-rec` in the theorem prover Vampire in order to easily and consistently extract the information described in (C1) from input function definitions. Also implement the keyword `match` to allow a more readable and robust way of defining case distinctions on function definition arguments. These are given in more detail in Chapter 5.

C4. *Extract function definition information from input axioms* in a way that user-error can be detected up to a reasonable level, including a well-foundedness and well-definedness check. Use the information for induction when the sufficient conditions for soundness are given. We present these concepts in detail in Chapter 5.

CHAPTER 2

Preliminaries

In this chapter we introduce the necessary underlying concepts needed for this thesis. First, a basic introduction of multi-sorted first-order logic with equality is presented to fix our notation. This will include some theory reasoning extensions required by induction and recursive functions. Then, the superposition calculus is shown on a higher level, after which we describe how saturation-based theorem proving works.

2.1 First-order logic with equality

We base our introduction to first-order logic with equality on an assumed basic familiarity with the topic. For a detailed introduction, see [BM10]. We define the *multi-sorted signature* as a set of *function* symbols $\mathcal{F}$ and a set of *predicate* symbols $\mathcal{P}$ together with a set of sorts $\mathcal{S}$. Each symbol is associated with an *arity* and a sort/type and a function with arity of 0 is called a *constant*. We also add $=$ to the language as a binary predicate symbol, $\top$ (true) and $\bot$ (false) as arity 0 predicate symbols and the unary function symbol *srt* mapping each term to its sort. We denote functions with letters f, g, h and predicates with letters p, q, r. We use from now on the words sort and type interchangeably.

Elements of the set of *variables* $\mathcal{V}$ are denoted by x, y, z, possibly with indices. The set of *terms* $\mathcal{T}$ is defined inductively using function symbols and variables. For terms, we use the letters s, t, etc. We reserve the notation σ, σ_0, σ_1, etc. for Skolem constants. We say that a term is *ground* if it contains no variables. We reserve the notation $\overline{x}$ and $\overline{t}$ for tuples of variables and terms, respectively. *Atoms* are built inductively from terms and predicate symbols. We use the unary logical connective $\neg$ for *negation*. Atoms and their negations are called *literals*. For a literal l, we use the notation $\overline{l}$ to denote its opposite sign literal. Moreover, we add binary logical connectives $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$ for *disjunction*, *conjunction*, *implication* and *equivalence*, respectively and the *universal quantifier* $\forall$ and the *existential quantifier* $\exists$. We use these connectives to build *formulas* from atoms. We

may abbreviate $\forall x.srt(x) = \alpha \to F$ as $\forall x \in \alpha.F$.

Additionally, we say that a set of literals is called a *clause* and treat it as a disjunction of the contained literals. We treat sets of clauses as conjunctions of the clauses. We reserve the symbol $\square$ for the *empty clause* which is logically equivalent to $\bot$. We call every term, literal, clause or formula an *expression*. We use the notation $s \trianglelefteq t$ to denote that s is a *subterm* of t and $s \triangleleft t$ if s is a *proper subterm* of t.

We distinguish special sorts called *inductive sorts/types*, function symbols for inductive sorts called *term constructors* and *term destructors* (or *constructors* and *destructors* for short). We require that the signature contains at least one constant constructor symbol for every inductive type. Such a symbol is called a *base constructor*, while non-constant ones are called *recursive constructors*. We call the ground terms built from the constructor symbols of a sort its *term algebra*.

Apart from the standard syntax, we introduce function symbols `ite`, `match` and `let`:

- `ite` is a ternary function symbol with type $\texttt{bool} \times \alpha \times \alpha \to \alpha$.
- `match` is an n+1-ary function symbol with type $\alpha_0 \times \texttt{pair}(\alpha_0, \alpha_1) \times ... \times \texttt{pair}(\alpha_0, \alpha_1) \to \alpha_1$. We also require that α_0 is an inductive type, the first argument is a known variable and the set of first elements of each pair of arguments $2 \leq j \leq n+1$ contains an *exhaustive* and *mutually exclusive* list of constructor terms of type α_0.
- `let` is a binary function symbol with type $\texttt{pair}(\alpha_0, \alpha_0) \times \alpha_1 \to \alpha_1$ with the first element of the first argument is required to be a variable.

We often use the more readable notation **if** t_1 **then** t_2 **else** t_3 instead of $\texttt{ite}(t_1, t_2, t_3)$, **match** x **with** $(s_1 \to t_1)...(s_n \to t_n)$ instead of $\texttt{match}(x, (s_1, t_1), ..., (s_n, t_n))$ and **let** $x = s$ **in** t instead of $\texttt{let}((x, s), t)$.

An *interpreted symbol* is a function or predicate whose meaning is defined through axioms, e.g. $=$ is an interpreted symbol in first-order logic with equality. All other symbols are called *uninterpreted*. In this thesis, we often work with interpreted function or predicate symbols defined through multiple axioms. In case of a function symbol `f`, we use the following notation for convenience:

$$\textbf{fun } \texttt{f}(x_1, ..., x_n) := body$$

We abuse this notation for predicate definitions, noting that strictly speaking these are not necessarily defined by equational axioms. Such definition for a predicate symbol `p` is as follows with a `bool` value:

$$\textbf{fun } \texttt{p}(x_1, ..., x_n) := body$$

This notation is mainly to distinguish function and predicate definitions needed for our purposes from all other axioms that contain the interpreted symbols.

Example 2.1.1 Some examples for function definitions are shown below. The identity function is:

$$\textbf{fun } \texttt{id}(x : \tau) \to \tau := x$$

A recursive function calculating the Fibonacci-numbers for natural numbers:

$$\begin{aligned}&\textbf{fun } \texttt{fib}(x : \texttt{nat}) \to \texttt{nat} :=\\ &\quad \textbf{if } x \leq \texttt{s}(\texttt{0}) \textbf{ then } x\\ &\quad \textbf{else } \texttt{fib}(\texttt{p}(x)) + \texttt{fib}(\texttt{p}(\texttt{p}(x)))\end{aligned}$$

Or an equivalent definition:

$$\begin{aligned}&\textbf{fun } \texttt{fib}(x : \texttt{nat}) \to \texttt{nat} :=\\ &\quad \textbf{match } x \textbf{ with}\\ &\quad\quad (\texttt{0} \to \texttt{0})\\ &\quad\quad (\texttt{s}(\texttt{0}) \to \texttt{s}(\texttt{0}))\\ &\quad\quad (\texttt{s}(\texttt{s}(z)) \to \texttt{fib}(\texttt{s}(z)) + \texttt{fib}(z))\end{aligned}$$

For the semantics, we use an *interpretation structure* $\mathcal{M}$ which incorporates a non-empty domain or *universe* $\mathcal{D}$ and an *interpretation function* $\mathcal{I}$ for variables, function symbols and predicate symbols. The interpretation of terms and formulas is then built up inductively from this. We additionally describe the semantics of our special functions:

- $\mathcal{I}(\texttt{ite}(s, t_1, t_2)) = \begin{cases}\mathcal{I}(t_1) & \text{if } \mathcal{I}(s) = \mathit{true}\\ \mathcal{I}(t_2) & \text{otherwise}\end{cases}$
- $\mathcal{I}(\texttt{match}(x, (s_1, t_1), ..., (s_n, t_n))) = \mathcal{I}(t_i[x/s_i])$, where $\mathcal{I}(x)$ matches $\mathcal{I}(s_i)$. This is well-defined as in the syntax part we require a complete and mutually exclusive list of terms for $s_1, .., s_n$.
- $\mathcal{I}(\texttt{let}((x, s), t) = \mathcal{I}(t[x/s])$

We note here that even though the semantics of `match` can be simulated by using `ite` and `let` expressions, we still find it very useful and compact to write case distinctions on term algebra terms.

An interpretation $\mathcal{M}$ is a *model* of a formula F denoted by $\mathcal{M} \models F$ if F evaluates to true in $\mathcal{M}$. Moreover, F is *satisfiable* if it has a model, otherwise if it has no model, it is *unsatisfiable*. F is *valid* if all interpretations are models of it, denoted by $\models F$.

We call a *substitution* θ a mapping of the form $\{s_1 \mapsto t_1, ..., s_n \mapsto t_n\}$ s.t. $s_i \neq s_j$ for all $1 \leq i, j \leq n$, $i \neq j$. An *application of a substitution* on an expression E is denoted by $E\theta$ and corresponds to the expression where all occurrences of s_i are replaced with t_i. Another notation is $E[s]$ where we mean there is a distinguished occurrence of the term s

in E. $E[\cdot]$ then means this particular occurrence of s is replaced by a *hole* and $E[t]$ that the occurrence is changed to a term t. We may use this notation also to denote multiple occurrences at the same time. A substitution θ is a *unifier* of two expressions E_1 and E_2 if $E_1\theta = E_2\theta$. A unifier θ is a *most general unifier* (mgu for short) if for all θ' unifiers of the same expressions, there is a $\theta^\star$ substitution s.t. $\theta' = \theta\theta^\star$.

A *position* is a finite sequence of positive integers, a *root position* is the empty sequence and is denoted by ϵ. We denote the concatenation of two position p and q by pq. We use positions in case of terms such that if a term $f(t_1, ..., t_n)$ is in position p, then term t_i is in position pi, for $1 \leq i \leq n$. We denote a subterm of term t in position p by $t|_p$. We extend positions to literals in the following way: a literal is always in position ϵ, moreover an equality $l = r$ has positions 1 and 2, corresponding to l and r, respectively and a predicate $p(s_1, ..., s_m)$ has positions 1 to m corresponding to s_1 to s_m, respectively.

A *binary relation* R on a set A is the subset of the Cartesian product $A \times A$. We usually write $a\ R\ b$ instead of $(a, b) \in R$ to denote that a is related to b w.r.t. R. Given two binary relations R_1 and R_2 on the same set A, $R_2 \cdot R_1$ denotes the relation s.t. for all pairs $(a, c) \in R_2 \cdot R_1$ with $a, c \in A$ there is some $b \in A$ where $(a, b) \in R_2$ and $(b, c) \in R_1$. An *irreflexive relation* R on A is such that $a\ R\ b$ implies $a \neq b$ for any $a, b \in A$. It holds for a *transitive relation* that for any $a, b, c \in A$, $a\ R\ b$ and $b\ R\ c$ entails $a\ R\ c$. A *proper order* (or *strict order*) is an irreflexive and transitive relation. A relation R on a set A is *well-founded* if every non-empty subset of A has at least one minimal element w.r.t. R.

A relation R on a set A is a *simplification ordering* if:

- it is *stable under substitutions*, i.e. $a\ R\ b$ implies $a\theta\ R\ b\theta$ for all $a, b \in A$ and substitutions θ
- it is *monotonic*, i.e. $a\ R\ b$ implies $s[a]\ R\ s[b]$ for all $a, b, s \in A$
- it has the *subterm property*, i.e. $a \lhd b$ implies $a\ R\ b$

2.1.1 Term algebras

Semantically, each n-ary constructor c has n corresponding destructors $d_1, ..., d_n$. For any constructor term $c(t_1, ..., t_n)$ with root symbol c, the following holds:

$$\forall 1 \leq i \leq n. d_i(c(t_1, ..., t_n)) = t_i$$

Moreover, we axiomatise every term algebra for an inductive type τ with constructors $c_1, ..., c_n$ of arities $m_1, ..., m_n$ and corresponding destructors $d_{11}, ..., d_{1m_1}, ..., d_{n1}, ..., d_{nm_n}$ as follows:

- The *term algebra injectivity axiom* states that if two constructor terms with identical top-level symbols c of arity n are equal, then their arguments are equal pairwise:

$$\forall x_1, ..., x_n, y_1, ..., y_n. c(x_1, ..., x_n) = c(y_1, ..., y_n) \rightarrow (x_1 = y_1 \wedge ... \wedge x_n = y_n)$$

- The *term algebra distinctness axiom* states that any two constructor terms with different top-level constructor symbols c_1 of arity n and c_2 of arity m are not equal:

$$\forall x_1, ..., x_n, y_1, ..., y_m . c_1(x_1, ..., x_n) \neq c_2(y_1, ..., y_m)$$

- The *term algebra exhaustiveness axiom* states that any term of type τ must equal to one of the constructors. In its most compact form, it also connects the constructors to their appropriate destructors:

$$\forall x \in \tau . x = c_1(d_{11}(x), ..., d_{1m_1}(x)) \vee ... \vee x = c_n(d_{n1}(x), ..., d_{nm_n}(x))$$

- The *term algebra acyclicity axiom* states that any term of type τ cannot be the subterm of itself. This can be formalized with the help of a predicate $\texttt{subterm}_\tau$. For each constructor c of arity n, we add the following axioms:

$$\forall x_1, ..., x_n . \texttt{subterm}_\tau(x_1, c(x_1, ..., x_n)) \wedge ... \wedge \texttt{subterm}_\tau(x_n, c(x_1, ..., x_n))$$

$$\forall y, x_1, ..., x_n . (\texttt{subterm}_\tau(y, x_1) \vee ... \vee \texttt{subterm}_\tau(y, x_n)) \rightarrow \texttt{subterm}_\tau(y, c(x_1, ..., x_n))$$

Then, the main axiom for acyclicity is as follows:

$$\forall x . \neg \texttt{subterm}_\tau(x, x)$$

Example 2.1.2 The natural numbers $\mathbb{N}$ can be represented as an inductive type

$$\texttt{nat} := \texttt{0} \mid \texttt{s}(x)$$

with base constructor `0`, unary recursive constructor `s` and corresponding destructor `p`. Some ground terms for natural numbers are `0` (~ 0), `s(0)` (~ 2), `p(s(0))` (~ 1), etc.

Polymorphic lists of type α (`list`(α)) can be formulated as

$$\texttt{list} := \texttt{nil} \mid \texttt{cons}(x, y)$$

with `nil` as base constructor and `cons` as binary recursive constructor with argument types α and `list`, and destructors `head` and `tail`. Some ground terms of `list`(`nat`) are `nil`, `cons`(`0`, `nil`), `cons`(`s`(`0`), `cons`(`0`, `nil`)).

Binary trees containing a value of type α are defined as:

$$\texttt{btree} := \texttt{leaf} \mid \texttt{node}(x, y, z)$$

where the first and third arguments of the `node` constructor are of type `btree` while the second is of type α.

From now on, we may use n instead of $\texttt{s}^n(\texttt{0})$ for ground terms of `nat` and $x_1 +\!\!+ ... +\!\!+ x_n$ instead of $\texttt{cons}(x_1, ...\texttt{cons}(x_n, \texttt{nil}))$ for terms of `list` to increase readability.

2.2 The superposition calculus

A theorem prover is provided with a set of axioms corresponding to the theory under consideration and a conjecture to be proven. Most theorem provers use some kind of *inference system* to automate the creation of new formulas or simplification of old ones. An *inference* is a rule of the form:

$$\frac{F_1 \quad ... \quad F_n}{G}$$

We say that $F_1, ..., F_n$ are the *premises* and G is the *conclusion* of the inference. Inferences with no premises are called *axioms*. If we can apply an inference rule because the premises can be matched with axioms or formulas already derived, then we instantiate the inference rule with the concrete premises and derive the conclusion, adding it to the set of derived formulas. A set of inference rules give an inference system.

An inference is a *simplifying* one if one or more of the premises can be removed from the set of formulas because they are redundant given the conclusion. We denote such a simplification with a crossed line:

$$\frac{\cancel{F_1} \quad ... \quad F_n}{G}$$

An inference rule is *sound* if its conclusion logically follows from its premises. An inference system is sound if all of its rule are. An inference system is *complete* if any valid set of formulas can be proven true in it.

Most modern first-order theorem provers use the *superposition calculus* (𝕊up) as their inference system. It works on sets of clauses, so all input formulas must be converted to *clausal normal form* (or CNF) before they can be used with the calculus. For our purposes, we will treat conversion to CNF as a black box and refer the reader to [RSV16]. We write $\texttt{cnf}(F)$ to denote the conversion of a formula F to its conjunctive normal form. 𝕊up is sound and *refutationally complete*. A *refutation* is a derivation of $\bot$. Refutational completeness means for any unsatisfiable formula set, we can derive the empty clause. Therefore, with superposition we usually negate our input conjecture and try to refute it which, if successful, means the original conjecture is valid.

For the completeness proof of 𝕊up, an ordering of terms, literals and clauses is necessary. In particular, we are interested in simplification orderings due to the following result [BN98]:

Theorem 2.2.1 Every simplification ordering is well-founded.

Widely-used simplification orders include *KBO* (Knuth-Bendix ordering) [KB70] and *LPO* (lexicographic path ordering) [KL80]. A discussion of these can be found in [Der82].

A first-order theorem prover uses a simplification term ordering $\succ$ as the basis to order terms, which is then extended to equalities, literals and clauses [BG94]. An equality with sides l and r is oriented as $l = r$ if $l \succ r$. We then use the *multiset extension* of the ordering to extend this from terms to literals and from literals to clauses. We will abuse notation and use the same symbol $\succ$ to denote the original ordering and its extensions.

We now present some of the inference rules necessary for this thesis from $\mathbb{S}$up:

Superposition rules are

$$\frac{l = r \vee C \qquad t[s] = u \vee D}{(t[r] = u \vee C \vee D)\theta}\,(\texttt{Sup}) \qquad \frac{l = r \vee C \qquad t[s] \neq u \vee D}{(t[r] \neq u \vee C \vee D)\theta}\,(\texttt{Sup})$$

$$\frac{l = r \vee C \qquad L[s] \vee D}{(L[r] \vee C \vee D)\theta}\,(\texttt{Sup})$$

where θ is an mgu of l and s, s is not a variable, $r\theta \not\succeq l\theta$, $u\theta \not\succeq t[s]\theta$ and L is not an equality literal.

Binary resolution is

$$\frac{A \vee C \qquad \neg B \vee D}{(C \vee D)\theta}\,(\texttt{Bin})$$

where θ is the mgu of A and B.

Equality resolution is

$$\frac{l \neq r \vee C}{C\theta}\,(\texttt{ER})$$

where θ is the mgu of l and r.

Moreover, we add a special case of superposition as a simplification rule, called *demodulation*, which is

$$\frac{l = r \qquad C[l\theta] \vee D}{C[r\theta] \vee D}\,(\texttt{Dem})$$

where $l\theta \succ r\theta$ and $C[l\theta] \vee D \succ l\theta = r\theta$.

2.3 Saturation-based proof search

Given a set of input formulas C in clausal form, the set of all derivable clauses using inference rules from $\mathbb{S}$up with the ordering $\succ$ from the set C is called the *closure of F w.r.t. $\mathbb{S}up$*. If the closure contains $\square$, the original set C is unsatisfiable, otherwise it is satisfiable. The process of computing the closure is called *saturation*. However, this process often results in using up an unlimited amount of time and memory, rendering it practically unusable. This is partly due to the undecidability of satisfiability for first-order theorems [G̈31] which means sometimes an infinite amount of resources is needed to prove a theorem. For that matter, in practice more subtle notions are needed to tackle this problem.

The first one is *saturation up to redundancy*. A clause C is *redundant* w.r.t a set of clauses S if some subset of S of clauses smaller than C w.r.t $\succ$ logically imply C. A clause can become redundant during saturation the moment it is added to the set of clauses or later when new clauses make it redundant. Clauses that never become redundant in the set of clauses during saturation are called *persistent*. Saturation up to redundancy means the closure is entirely made up of persistent clauses, redundant ones are ideally eliminated the moment they become redundant (or never added in the first place). One way of removing such clauses is adding *deletion rules* to the inference system such as tautology elimination. Redundancy is in general undecidable, so we cannot eliminate all redundant clauses in practice. We use cheap simplification rules instead.

Other methods of controlling the search space and the saturation process are selection methods which control the order in which the inferences are applied. For a more detailed discussion on saturation algorithms see [KV13].

CHAPTER 3

Induction schemes and proofs

In this chapter we look at why induction proofs are necessary to prove some formulas and how they are carried out manually. During this, we will note some key steps of these proofs which are indispensable for a successful automated inductive proof. Then, we formally introduce the mathematical basis of well-founded induction and show how this notion is adapted for proving conjectures with inductive types in a saturation-based theorem prover. Finally, we look at the various technical challenges that arise when proving more difficult formulas.

3.1 Induction from a mathematician's perspective

Inductive reasoning proved to be a handy tool in the mathematician's repertoire throughout the centuries. The technique's power lies in the fact that it can prove statements about *infinite* structures with only a *finite* amount of reasoning. This is therefore essential in all inductive theories. However, a technique so powerful comes with its limitations and one may wonder: why is it so hard to automate?

A hint of answer may be found in the interconnection between recursion, induction and well-foundedness. Seen through Gödel's first incompleteness theorem [Gö31], every non-trivial inductive theory (such as integer arithmetic) has some formulas that cannot be proven true. From another viewpoint, deciding whether a recursive function is well-founded, i.e. whether it terminates is equivalent to the Halting Problem and is undecidable [Tur37]. Therefore, for any such recursive function, we cannot possibly select a corresponding single well-founded relation, which we will later see is equivalent to selecting an induction formula.

Consider now the following conjecture – the associativity of addition over natural numbers – which demonstrates some of the difficulties of inductive reasoning:

$$\forall x, y, z.\mathtt{add}(\mathtt{add}(x, y), z) = \mathtt{add}(x, \mathtt{add}(y, z))$$

We also need the definition of add:

$$\begin{array}{l}\textbf{fun } \mathtt{add}(x : \mathtt{nat}, y : \mathtt{nat}) \to \mathtt{nat} := \\ \quad \textbf{match } x \textbf{ with} \\ \quad\quad (\mathtt{0} \to y) \\ \quad\quad (\mathtt{s}(z) \to \mathtt{s}(\mathtt{add}(z, y)))\end{array}$$

The first thing a mathematician does is to select an appropriate induction formula, which needs the selection of an induction term – x, y or z – and selecting a well-founded relation. The induction formula in this case is the following:

$$\forall y, z. \left(\left(\begin{array}{c} \mathtt{add}(\mathtt{add}(\mathtt{0}, y), z) = \mathtt{add}(\mathtt{0}, \mathtt{add}(y, z)) \wedge \\ \forall x_0. \left(\begin{array}{c} \mathtt{add}(\mathtt{add}(x_0, y), z) = \mathtt{add}(x_0, \mathtt{add}(y, z)) \to \\ \mathtt{add}(\mathtt{add}(\mathtt{s}(x_0), y), z) = \mathtt{add}(\mathtt{s}(x_0), \mathtt{add}(y, z)) \end{array} \right) \\ \to \forall x. \mathtt{add}(\mathtt{add}(x, y), z) = \mathtt{add}(x, \mathtt{add}(y, z)) \end{array} \right) \right)$$

Let us now focus on the first part of the antecedent, the *base case*:

$$\mathtt{add}(\mathtt{add}(\mathtt{0}, y), z) = \mathtt{add}(\mathtt{0}, \mathtt{add}(y, z))$$

By using the definition of add, we can reduce this to:

$$\mathtt{add}(y, z) = \mathtt{add}(y, z)$$

This is already a tautology, so we can move on to the next part, the so-called *step case*:

$$\mathtt{add}(\mathtt{add}(x_0, y), z) = \mathtt{add}(x_0, \mathtt{add}(y, z)) \to \mathtt{add}(\mathtt{add}(\mathtt{s}(x_0), y), z) = \mathtt{add}(\mathtt{s}(x_0), \mathtt{add}(y, z))$$

The implication conclusion can be simplified by the definition of add, twice on the left-hand side, once on the right-hand side:

$$\mathtt{add}(\mathtt{add}(\mathtt{s}\,(\,x_0\,), y), z) = \mathtt{add}(\mathtt{s}\,(\,x_0\,), \mathtt{add}(y, z))$$

$$\mathtt{add}(\mathtt{s}\,(\,\mathtt{add}(x_0, y)\,), z) = \mathtt{s}\,(\,\mathtt{add}(x_0, \mathtt{add}(y, z))\,)$$

$$\mathtt{s}\,(\,\mathtt{add}(\mathtt{add}(x_0, y), z)\,) = \mathtt{s}\,(\,\mathtt{add}(x_0, \mathtt{add}(y, z))\,)$$

Notice how the structure changes in each simplification stage, in a way that the s parts "bubble up" and how it ends with an equality which contains inside the s terms exactly the two sides of the *induction hypothesis*, that is, the antecedent of the inductive step case. So after simplification, the original implication becomes:

$$\mathtt{add}(\mathtt{add}(x_0, y), z) = \mathtt{add}(x_0, \mathtt{add}(y, z)) \to \mathtt{s}(\mathtt{add}(\mathtt{add}(x_0, y), z)) = \mathtt{s}(\mathtt{add}(x_0, \mathtt{add}(y, z)))$$

This can be proven in several ways. E.g. by invoking the injectivity axiom of the term algebra constructor s:

$$\forall x, y. \mathtt{s}(x) = \mathtt{s}(y) \to x = y$$

After this we get a tautology:

$$\texttt{add}(\texttt{add}(x_0, y), z) = \texttt{add}(x_0, \texttt{add}(y, z)) \rightarrow \texttt{add}(\texttt{add}(x_0, y), z) = \texttt{add}(x_0, \texttt{add}(y, z))$$

Another technique we can use is called *equality resolution* (which is also called *cross-fertilization* in the literature [BM79]) which is essentially rewriting with the hypothesis and then throwing it away, which results once again in a tautology.

Since both conjuncts of the implicant in the original induction formula are true, we can conclude that $\forall x, y, z.\texttt{add}(\texttt{add}(x, y), z) = \texttt{add}(x, \texttt{add}(y, z))$ is also true, which is what we are trying to prove.

This induction formula succeeded because of the "right" choice of:

1. the *induction term* (in this case x) which allowed us to use the recursive definition of `add` in a way that we could get rid of the initial `s` on both sides in the step case and reduce the base case to a tautology
2. the *case distinction* which provided us with an easily provable base case and a step case in which the simplified implicant could be rewritten using the induction hypothesis

In general, both choices give a branching point which may cause a blow-up in the search space. Later, we will address these issues (and some others that did not come up in this simple example) and develop heuristics that makes the right choices for certain types of problems.

3.2 Well-founded induction

Let us first give the theoretical basis for the structure of the induction formula, that is, let us introduce well-founded induction. Well-founded orders give the basis of both terminating recursive functions and valid induction formulas. Now we give the most general form of induction which is based on well-founded orders.

Definition 3.2.1 Let A be a set, $\prec$ a well-founded order over A and $P[x]$ a first-order statement with x as a free variable denoting an element of A. We define the **well-founded** (or **Noetherian**) **induction scheme** as:

$$\Big(\forall u \in A.(\forall v \in A.(v \prec u \rightarrow P[v]) \rightarrow P[u])\Big) \rightarrow \forall w \in A.P[w]$$

In words: for all elements $u \in A$, if for all smaller elements $v \in A$ (w.r.t. $\prec$) $P[v]$ holds implies that $P[u]$ holds, then P holds for all elements in A.

Because the minimal elements have no smaller elements w.r.t. $\prec$, this means that for them the premise is vacuously true. For our purposes, it is sufficient to adapt this definition to the term algebra types. We choose simplification orderings for single elements of any term algebra, focusing on the subterm property which is easy to check. We can then extend these orderings to tuples of elements by taking e.g. the lexicographical extension of the atomic relations. We discuss exactly what orderings will be used in this thesis later.

By adjusting how many subterms a certain term is in relation with, we can create many induction schemes with different such well-founded orders and stronger or weaker induction steps.

Example 3.2.1 Some induction schemes for `nat` are:

- only considering immediate subterms (i.e. $\forall x.x \prec \mathtt{s}(x)$):

$$(P[\mathtt{0}] \land \forall x.(P[x] \to P[\mathtt{s}(x)])) \to \forall z.P[z]$$

- considering the immediate subterms of immediate subterms (i.e. $\forall x.x \prec \mathtt{s}(\mathtt{s}(x))$):

$$(P[\mathtt{0}] \land P[\mathtt{s}(\mathtt{0})] \land \forall x.(P[x] \to P[\mathtt{s}(\mathtt{s}(x))])) \to \forall z.P[z]$$

- considering every subterm:

$$(P[\mathtt{0}] \land \forall x.\forall y \lhd x.(P[y] \to P[x])) \to \forall z.P[z]$$

The last induction scheme in particular corresponds to *strong mathematical induction* over the natural numbers.

3.3 Duality between recursion and induction

So what is the connection in the previous example between the correct induction formula and the formula we want to prove? We can notice the similarity between the case distinction and recursive calls in the definition of `add` and the induction base and step cases. To make the connection, first we simplify things by introducing *relational descriptions* based on the notation of Walther [Wal92].

Definition 3.3.1 A **relational description** (or **r-description** for short) for an n-ary function `f` is a triple:

$$(\mathtt{f}(t_1, ..., t_n), \{\mathtt{f}(t_{1i}, ..., t_{ni}) \mid 1 \leq i \leq m\}, F)$$

where $m \geq 0$ and all t_j and t_{ji} ($1 \leq j \leq n$ and $1 \leq i \leq m$) are constructor terms containing only constructor symbols or variables and F is a set of formulas, the **side conditions** of the r-description.

An r-description contains the necessary information for induction for a single branch of a function definition. We can create r-descriptions for a non-recursive function although induction on statements with such functions is not really necessary.

Example 3.3.1 For `id`, the r-descriptions are:

$$\{(\texttt{id}(x), \emptyset, \emptyset)\}$$

For `fib`, these are:

$$\begin{Bmatrix} (\texttt{fib}(x), & \emptyset, & \{x \leq \texttt{s}(\texttt{0})\}), \\ (\texttt{fib}(x), & \{\texttt{fib}(\texttt{p}(x)), \texttt{fib}(\texttt{p}(\texttt{p}(x)))\}, & \{x > \texttt{s}(\texttt{0})\}) \end{Bmatrix}$$

or

$$\begin{Bmatrix} (\texttt{fib}(\texttt{0}), & \emptyset, & \emptyset), \\ (\texttt{fib}(\texttt{s}(\texttt{0})), & \emptyset, & \emptyset), \\ (\texttt{fib}(\texttt{s}(\texttt{s}(z))), & \{\texttt{fib}(\texttt{s}(z)), \texttt{fib}(z)\}, & \emptyset) \end{Bmatrix}$$

depending which definition we use.

Next, we need to differentiate between arguments that take part in the case distinction from the ones that do not.

Definition 3.3.2 Given a function `f` of arity n and a corresponding r-description

$$R = (\texttt{f}(t_1, ..., t_n), \{\texttt{f}(t_{1j}, ..., t_{nj}) \mid 1 \leq j \leq m\}, F)$$

an argument position $1 \leq i \leq n$ is an **active argument position of** R if there is a $1 \leq j \leq m$ with $t_i \neq t_{ij}$ and at least one of t_i and t_{ij} is a non-variable. We call an argument position an **active argument position of** `f` if it is an active argument position of some r-description of `f`. We denote the set of active argument positions of `f` with $I^a_{\texttt{f}}$.

One example for a function with multiple active argument positions is the Ackermann-function [Ack28] shown in Figure 3.1. The r-descriptions corresponding to this function are:

$$\begin{Bmatrix} (\texttt{ack}(\texttt{s}(z), \texttt{0}), & \{\texttt{ack}(z, \texttt{s}(\texttt{0}))\}, & \emptyset) \\ (\texttt{ack}(\texttt{s}(z), \texttt{s}(w)), & \{\texttt{ack}(z, \texttt{ack}(\texttt{s}(z), w)), \texttt{ack}(\texttt{s}(z), w)\}, & \emptyset) \end{Bmatrix}$$

This function is known to produce numbers of extraordinary size but its termination is nonetheless easy to show. In fact, any terminating (recursive) function suggests a

$$
\begin{aligned}
&\textbf{fun } \mathtt{ack}(x : \mathtt{nat}, y : \mathtt{nat}) \to \mathtt{nat} := \\
&\quad \textbf{match } x \textbf{ with} \\
&\qquad (\mathtt{0} \to \mathtt{s}(y)) \\
&\qquad (\mathtt{s}(z) \to \textbf{match } y \textbf{ with} \\
&\qquad\quad (\mathtt{0} \to \mathtt{ack}(z, \mathtt{s}(\mathtt{0}))) \\
&\qquad\quad (\mathtt{s}(w) \to \mathtt{ack}(z, \mathtt{ack}(\mathtt{s}(z), w))))
\end{aligned}
$$

Figure 3.1: The Ackermann-function

well-founded order $\prec$. We are interested in particular in lexicographic extensions of well-founded orders:

Definition 3.3.3 Let $\prec_1$ and $\prec_2$ be two orders defined on sets A_1 and A_2. The **lexicographic order** of these two orders is denoted by $(\prec_1, \prec_2)_{lex}$ and is defined on the product $A_1 \times A_2$ as follows: $(a_1, a_2)(\prec_1, \prec_2)_{lex}(b_1, b_2)$ if and only if, (i) $a_1 \prec_1 b_1$ or (ii) $a_1 = b_1$ and $a_2 \prec_2 b_2$.

The following result states the well-foundedness of lexicographic extensions [Mid]:

Theorem 3.3.1 The lexicographic order defined by two well-founded orders is well-founded.

The definition and result generalizes to lexicographic orders defined by more than two orders analogously. We now define sufficient conditions for the termination of recursive functions based on well-founded lexicographic orders. In particular, we consider only the set of active argument positions of a function, since other positions do not change by definition and therefore cannot contribute to the order. We define two distinguished types for these indices with respect to each relation between function headers and their recursive calls.

Definition 3.3.4 Given a function header $\mathtt{f}(t_1, ..., t_n)$ and a recursive call $\mathtt{f}(s_1, ..., s_n)$, an argument position $1 \leq i \leq n$ is a **fixed position** if $t_i = s_i$ and a **subterm position** if $t_i \triangleright s_i$.

Now let us group together sets of indices that can be used together in the lexicographic order:

Definition 3.3.5 For an r-description set $\{(\mathtt{f}(\overline{t_i}), \{\mathtt{f}(\overline{t_{ij}}) \mid 1 \leq j \leq m\}, F_i) \mid 1 \leq i \leq n\}$, an **ordered separation** of $I^a_{\mathtt{f}}$ is a tuple of pairwise distinct, exhaustive and

non-empty sets of indices $(I_1, ..., I_N)$, i.e. $\forall k.\forall l \neq k.I_k \cap I_l = \emptyset$ and $\cup_k I_k = I^a_{\mathtt{f}}$.

Each of these sets contain the maximum number of indices such that in any pair $\mathtt{f}(\bar{t}_i)$ and $\mathtt{f}(\bar{t}_{ij})$ it holds that there is some $1 \leq k \leq N$ such that all $l < k$ we have $\forall l' \in I_l.t_{il'} = t_{ijl'}$ and we have $\forall k' \in I_k.t_{ik'} \triangleright t_{ijk'}$.

This makes the creation of a lexicographic order easier as any order inside one of these sets obeys the lexicographic definition. We proceed by stating a property of the ordered separation.

Lemma 3.3.1 If an ordered separation S exists for a function $\mathtt{f}$ with r-descriptions $\mathcal{R}$ and a non-empty $I^a_{\mathtt{f}}$, then $\mathtt{f}$ is terminating.

Proof. Suppose S exists but $\mathtt{f}$ is not terminating. This means that there is an infinite sequence of n-tuples of ground terms $T_0, T_1, ...$ collecting the arguments of an infinite recursive call chain of the n-ary $\mathtt{f}$. Since $I^a_{\mathtt{f}}$ is non-empty, there is at least one set in S, let us consider the first, I_1. For any $i \geq 0$, the elements of T_{i+1} at indices of I_1 by construction are either identical to the elements of T_i in the same indices or they are strict subterms of those. The latter cannot occur infinitely often, so we look at the former case. This can only happen if there is a second set I_2 in S. With the same reasoning on I_2, we get the same and move on to I_3 if it exists. Considering eventually all sets in S, and since no other indices change in any recursive calls outside of S, we get a contradiction. □

The converse is not necessarily true as we only consider orderings with the subterm property. Such a separation need not be unique either, e.g. a function with one recursive branch with recursive calls $(\mathtt{s}(x), y)$ and $(x, \mathtt{s}(y))$ for arguments $(\mathtt{s}(x), \mathtt{s}(y))$ has two ordered separations, $(\{1\}, \{2\})$ and $(\{2\}, \{1\})$.

In case of `ack`, the well-founded relation is simply:

- $(z, \mathtt{s}(\mathtt{0})) \prec_{\mathtt{ack}} (\mathtt{s}(z), \mathtt{0})$ due to $\mathtt{s}(z) \triangleright z$
- $(z, \mathtt{ack}(\mathtt{s}(z), w)) \prec_{\mathtt{ack}} (\mathtt{s}(z), \mathtt{s}(w))$ due to $\mathtt{s}(z) \triangleright z$
- $(\mathtt{s}(z), w) \prec_{\mathtt{ack}} (\mathtt{s}(z), \mathtt{s}(w))$ due to $\mathtt{s}(z) = \mathtt{s}(z)$ and $\mathtt{s}(w) \triangleright w$

The above can be extracted from the r-description set as follows:

1. We take the set of active argument positions $I^a_{\mathtt{ack}} = \{1, 2\}$
2. We then separate these into sets such that they "change together" in all relations from some terms to one of their subterms, or remain fixed together:
   - In the first relation between $\mathtt{ack}(\mathtt{s}(z), \mathtt{0})$ and $\mathtt{ack}(z, \mathtt{s}(\mathtt{0}))$, only the first position is a subterm position, this already separates the two into $\{\{1\}, \{2\}\}$.

$$
\begin{array}{l}
\textbf{fun}\ \mathtt{leq}(x : \mathtt{nat}, y : \mathtt{nat}) \to \mathtt{bool} := \\
\quad \textbf{match}\ x\ \textbf{with} \\
\quad\quad (\mathtt{0} \to \mathtt{true}) \\
\quad\quad (\mathtt{s}(z) \to \textbf{match}\ y\ \textbf{with} \\
\quad\quad\quad (\mathtt{0} \to \mathtt{false}) \\
\quad\quad\quad (\mathtt{s}(w) \to \mathtt{leq}(z, w)))
\end{array}
$$

Figure 3.2: The recursive function for $\leq$

- In the second relation between $\mathtt{ack}(\mathtt{s}(z), \mathtt{s}(w))$ and $\mathtt{ack}(z, \mathtt{ack}(\mathtt{s}(z), w)), \mathtt{ack}(\mathtt{s}(z), w)$, the first position is a subterm. The second position changes but is not a subterm, so we may separate them the same as in the previous relation.
- Finally, in the third relation between $\mathtt{ack}(\mathtt{s}(z), \mathtt{s}(w))$ and $\mathtt{ack}(\mathtt{s}(z), w)$, the first argument position is fixed, the second position is a subterm position, so it gives the same result as the first.

3. According to the first and second relations, we can identify position 1 as the first in the order. Based on the third relation, position 2 is the second in the order, since when it changes position 1 is fixed. Therefore, this separation $\{\{1\}, \{2\}\}$ can be ordered as $(\{1\}, \{2\})$ or simply $(1, 2)$ to get the well-founded order given above.

Other functions may need different kinds of well-founded relations, e.g. the recursive definition of $\leq$ for natural numbers shown in Figure 3.2. The termination of this function is based on the well-founded relation $(x, y) \prec (\mathtt{s}(x), \mathtt{s}(y))$ and this can be calculated from the only relation where $\mathtt{leq}(\mathtt{s}(z), \mathtt{s}(w))$ changes to $\mathtt{leq}(z, w)$, giving the 1-tuple ordered separation $(\{1, 2\})$.

3.4 Induction schemes

So far we have seen how a particular function gives a set of r-descriptions which define a well-founded order over the active argument positions of the function arguments, and together they give an induction template.

Definition 3.4.1 An **induction template** for a function $\mathtt{f}$ is a set of r-descriptions and an ordered separation on their active arguments $I^a_{\mathtt{f}}$.

Now we show how this template can be converted into a so-called induction scheme.

Definition 3.4.2 Given two tuples of terms $S := (s_1, ..., s_m)$ and $T := (t_1, ..., t_m)$ with $m > 0$, we define **a substitution from S to T** as:

$$\theta = \{s \mapsto t \mid \exists J \subseteq \{1, ..., m\}.J \neq \emptyset \wedge \forall j \in J.(s = s_j \wedge t = t_j \cdot mgu(\{t_j \mid j \in J\}))\}$$

This substitution – if it exists – essentially collects the identical terms of S and maps them to the unification of terms corresponding to the terms in the same indices in T. The substitution exists if it is well-defined, that is, if all elements in T corresponding to the same term from S are unifiable.

Definition 3.4.3 A function term $\mathtt{f}(s_1, ..., s_n)$ with a tuple of selected terms $(s'_{i_1}, ..., s'_{i_m})$ s.t. for all $1 \leq j \leq m$, $i_j \in I^a_{\mathtt{f}}$ and $s'_{i_j} \trianglelefteq s_{i_j}$, and an r-description

$$(\mathtt{f}(t_1, ..., t_n), \{\mathtt{f}(t_{1j}, ..., t_{nj}) \mid 1 \leq j \leq m\}, F)$$

together define an **r-description instance**

$$(\mu, \{\mu_j \mid 1 \leq j \leq m\}, F')$$

where μ and μ_j are non-empty and well-defined substitutions from $(s'_{i_1}, ..., s'_{i_m})$ to tuples $(t_{i_1}, ..., t_{i_m})$ and $(t_{i_1 j}, ..., t_{i_m j})$, respectively. Furthermore, $F' := \cup_{f \in F} f\theta$ where $\theta := \{t_i \mapsto s_i \mid i \in \{1, ..., n\} \setminus I^a_{\mathtt{f}}\}$, that is, the substitution of non-active arguments of the function term into the side conditions.

We call $dom(\mu)$ the **induction terms** of the r-description instance.

Definition 3.4.4 The set of all r-description instances defined by a function term $\mathtt{f}(s_1, ..., s_n)$ and the r-descriptions given by the induction termplate for $\mathtt{f}$ give an **induction scheme for $\mathtt{f}$**.

Moreover, we define **the recursive cases of an induction scheme** I (denoted by I^{rec}) as all r-description instances of I where the second tuple element is non-empty:

$$I^{rec} := \{(\mu, \{\mu_j \mid 1 \leq j \leq n\}, F) \mid n > 0\} \subseteq I$$

That is, for a function term we select a subterm in each of its active argument positions and create a mapping from these to the arguments in the same positions of the r-description function headers. This is only possible if the substitution is well-defined which is also ensured by the unification of the arguments if two or more of the selected terms are identical. Notice also that the r-description instance exists only if the main substitution μ is well-defined. On the other hand, we can manage without μ_js from recursive calls that are not well-defined, since by discarding these we will only lose induction hypotheses, still getting a valid induction formula.

We illustrate this with an example. The induction template of `leq` contains 3 r-description instances, each corresponding to a branch in the definition of `leq`:

$$\left\{\begin{array}{lll} (\mathtt{leq}(\mathtt{0}, y), & \emptyset, & \emptyset), \\ (\mathtt{leq}(\mathtt{s}(z), \mathtt{0}), & \emptyset, & \emptyset), \\ (\mathtt{leq}(\mathtt{s}(z), \mathtt{s}(w)), & \{\mathtt{leq}(z, w)\}, & \emptyset) \end{array}\right\} \tag{3.1}$$

Suppose we have a term $\mathtt{leq}(x, x)$. To generate an induction scheme from it, first we create a tuple of selected terms in active argument positions. Without much choice, we create the tuple (x, x). We look at the r-descriptions and try to create an r-description instance from each:

- For the first, we have to match the selected terms (x, x) with the terms in the same indices from $\mathtt{leq}(\mathtt{0}, y)$ which boils down to substituting `0` and y at the same time for x. Since y is a variable, they unify to `0` and the r-description instance is $(\{x \mapsto \mathtt{0}\}, \emptyset, \emptyset)$.
- In the second, we match (x, x) with both arguments of $\mathtt{leq}(\mathtt{s}(z), \mathtt{0})$. $\mathtt{s}(z)$ and `0` do not unify, so we have to discard this case.
- For the third r-description, we match the tuple with $(\mathtt{s}(z), \mathtt{s}(w))$. Since the two `s` terms unify, we can create the main substitution $\{x \mapsto \mathtt{s}(z)\}$. Next, we have to do the same with the recursive call terms. That is, we have to match (x, x) with the arguments of $\mathtt{leq}(z, w)$. The r-description instance is $(\{x \mapsto \mathtt{s}(z)\}, \{x \mapsto z\}, \emptyset)$.

The final induction scheme is the set of the two generated r-description instances:

$$\left\{\begin{array}{lll} (\{x \mapsto \mathtt{0}\}, & \emptyset, & \emptyset), \\ (\{x \mapsto \mathtt{s}(z)\}, & \{x \mapsto z\}, & \emptyset) \end{array}\right\}$$

An inductive goal we want to prove may give rise to many induction schemes. It can be the case that a term contains subterms in active argument positions that also generate induction schemes, e.g. $\mathtt{add}(\mathtt{add}(x, y), z)$ has an inner term $\mathtt{add}(x, y)$ which itself gives an induction scheme. This can be generated using the induction template of `add` where the first argument position is active:

$$\left\{\begin{array}{lll} (\mathtt{add}(\mathtt{0}, y), & \emptyset, & \emptyset) \\ (\mathtt{add}(\mathtt{s}(x_0), y), & \{\mathtt{add}(x_0, y)\}, & \emptyset) \end{array}\right\} \tag{3.2}$$

Then, the induction scheme for $\mathtt{add}(x, y)$ is the following:

$$\left\{\begin{array}{lll} (\{x \mapsto \mathtt{0}\}, & \emptyset, & \emptyset) \\ (\{x \mapsto \mathtt{s}(x_0)\}, & \{\{x \mapsto x_0\}\}, & \emptyset) \end{array}\right\}$$

As the inner term gives an induction scheme, we can use this instead of generating one for the outer term. For a more detailed discussion of selecting induction terms and occurrences, see Section 3.11.

We extend the definition of induction terms to that of an induction scheme by taking the union of those over the r-description instances of the scheme. We give later a more detailed description on how to use the ordered separation to get better induction schemes.

3.4.1 Generating an induction formula from an induction scheme

Using induction schemes before actually creating an *induction formula* is preferable as all three parts of an r-description instance are easily readable and modifiable this way without the need to parse subformulas. However, eventually we must convert the usable induction schemes to induction formulas and hand them to a first-order theorem prover. For this, we also need the original inductive goal or any other formula we want to perform induction upon. Without them, the induction scheme is just a skeleton. Given a formula $P[x_1]...[x_n]$ we induct on and an induction scheme with induction terms $x_1, ..., x_n$, we create a corresponding induction formula as follows:

1. for each substitution $\mu := \{x_i \mapsto t_i \mid 1 \leq i \leq n\}$, we create a formula $P_\mu := P[t_1]...[t_n]$
2. for each r-description instance $R := (\mu, \{\mu_j \mid 1 \leq j \leq m\}, F)$ with free variables $y_1, ..., y_l$ from substituted terms in μ and free variables $z_1, ..., z_k$ from substituted terms and formulas in μ_j and F, we generate the formula:
$$P_R := \forall y_1, ..., y_l.(\forall z_1, ..., z_k. \bigwedge_{f \in F} f \wedge \bigwedge_{1 \leq j \leq m} P_{\mu_j}) \rightarrow P_\mu$$
3. the final induction formula for r-description instances $R_1, ..., R_n$ with remaining free variables $u_1, ..., u_r$ of P (excluding $x_1, ..., x_n$) is then:
$$\forall u_1, ..., u_r. \bigwedge_{1 \leq i \leq n} P_{R_i} \rightarrow \forall x_1, ..., x_n.P[x_1]...[x_n]$$

Example 3.4.1 Suppose we have the following conjecture:

$$\forall x, y.\texttt{leq}(x, \texttt{add}(x, y))$$

This will generate the r-description instances based on $I^a_{\texttt{leq}} = \{1, 2\}$ and selected terms (x, x) using the r-description set 3.1 similar to the previous examples:

$$\left\{ \begin{array}{lll} R_1 := (\mu_1 := \{x \mapsto \texttt{0}\}, & \emptyset, & \emptyset), \\ R_2 := (\mu_2 := \{x \mapsto \texttt{s}(z)\}, & \mu_3 := \{\{x \mapsto z\}\}, & \emptyset) \end{array} \right\}$$

First, we generate the base formulas $P_{\mu_1}, P_{\mu_2}, P_{\mu_3}$:

$$P_{\mu_1} := \texttt{leq}(\texttt{0}, \texttt{add}(\texttt{0}, y))$$

$$P_{\mu_2} := \texttt{leq}(\texttt{s}(z), \texttt{add}(\texttt{s}(z), y))$$

$$P_{\mu_3} := \texttt{leq}(z, \texttt{add}(z, y))$$

Then, we generate the formulas P_{R_1}, P_{R_2}:

$$P_{R_1} := \texttt{leq}(\texttt{0}, \texttt{add}(\texttt{0}, y))$$

$$P_{R_1} := \forall z.\texttt{leq}(z, \texttt{add}(z, y)) \rightarrow \texttt{leq}(\texttt{s}(z), \texttt{add}(\texttt{s}(z), y))$$

Finally, we create the induction formula:

$$\forall y. \left(\begin{array}{c} \texttt{leq}(\texttt{0}, \texttt{add}(\texttt{0}, y)) \wedge \\ \forall z. \left(\begin{array}{c} \texttt{leq}(z, \texttt{add}(z, y)) \rightarrow \\ \texttt{leq}(\texttt{s}(z), \texttt{add}(\texttt{s}(z), y)) \end{array} \right) \end{array} \right) \rightarrow \forall x.\texttt{leq}(x, \texttt{add}(x, y))$$

3.5 Correspondence between well-founded orders and induction schemes

Our induction scheme definition contains all necessary information to generate induction formulas, however, there are certain mathematical definitions that are easier to understand and work with in terms of well-founded orders. Since on the level of induction schemes we do not have the ordered separation from the original induction templates, we instead create a new relation.

To this end, for an induction scheme $I := \{R_i := (\mu_i, \{\mu_{ij} \mid 1 \leq j \leq m_i\}, F_i) \mid 1 \leq i \leq n\}$, we can create a well-founded order $\prec_I$ as follows: we fix an order for the elements of $\bigcup_{1 \leq i \leq n} dom(\mu_i)$ for all $1 \leq i \leq m$, resulting in tuples of the form $(t_1, ..., t_k)$. For every (μ_i, μ_{ij}) pair, $(t_1\mu_{ij}, ..., t_k\mu_{ij}) \prec_I (t_1\mu_i, ..., t_k\mu_i)$ with side conditions F_i is in the relation.

From now on, we abuse the notation and use induction schemes, induction formulas and well-founded orders interchangably. Let us look at an example.

Example 3.5.1 The previous induction scheme for `leq`

$$\left\{ \begin{array}{ccc} (\{x \mapsto \texttt{0}\}, & \emptyset, & \emptyset), \\ (\{x \mapsto \texttt{s}(z)\}, & \{\{x \mapsto z\}\}, & \emptyset) \end{array} \right\}$$

with fixed order (x) gives rise to the well-founded order $(z) \prec (\texttt{s}(z))$.

3.6 Containment of induction schemes

Depending on the recursive calls a function definition contains, an induction scheme stemming from this function definition can define a well-founded order that is entirely contained in another one. Such a well-founded order is in a sense less useful than the other

because the latter makes the same case distinction and gives more induction hypotheses for each step case.

For an example, consider the type `btree` from Example 2.1.2 where we have multiple recursive substructures for a non-leaf node, namely `left` and `right`. One function may recurse only into one or the other in its recursive call, while others recurse into both branches. An induction formula

$$\begin{pmatrix} P[\texttt{leaf}] \wedge \\ \forall x, y, z.((P[x] \wedge P[z]) \rightarrow P[\texttt{node}(x, y, z)]) \end{pmatrix} \rightarrow \forall u.P[u] \tag{3.3}$$

can be used whenever another one

$$\begin{pmatrix} P[\texttt{leaf}] \wedge \\ \forall x, y, z.(P[x] \rightarrow P[\texttt{node}(x, y, z)]) \end{pmatrix} \rightarrow \forall u.P[u] \tag{3.4}$$

is used. The reason for this is that the formula (3.3) defines *stronger induction hypotheses* $(P[x] \wedge P[z])$ than the formula (3.4) (only $P[x]$) and therefore, since this weakens the inductive step case and the whole induction formula antecedent, formula (3.3) is weaker than (3.4), in fact, the first is subsumed by the second. This means that even though both formulas are based on well-founded relations and are valid, the inductive step case in the second will be false for more properties P and thus these properties cannot possibly be proven with the second formula.

In terms of well-founded orders, we can formally define this property:

Definition 3.6.1 Let $\prec_1$ and $\prec_2$ two well-founded orders over elements of a set A. $\prec_1$ **is contained by** $\prec_2$ whenever we have $x \prec_1 y$ for all x, y, then $x \prec_2 y$. Concisely written:

$$\prec_1 \subseteq \prec_2$$

For two well-founded orders $\prec_1, \prec_2$ over a type τ, if $\prec_1 \subseteq \prec_2$, we can safely use $\prec_2$ in an induction formula instead of $\prec_1$ as it uses the same case distinction as $\prec_1$ and for each case gives at least as many induction hypotheses as the other and maybe more. From a different viewpoint, $\prec_2$ relates more pairs than $\prec_1$ and with more pairs, the chances are higher that we can use the left-hand sides of these pairs to prove the right-hand sides.

As a rule of thumb, we want to store as much information in the final induction formula about the functions in "good" positions in the current inductive goal as possible to ensure that all such functions will match the terms in the induction step. For example, consider the variant `add2` for `add` shown in Figure 3.3.

The well-founded order $\prec_{\texttt{add2}}$ on the first argument is the same as in the case of `add` except that it starts at $\texttt{s}(\texttt{0})$ instead of $\texttt{0}$:

$$\texttt{s}(\texttt{0}) \prec_{\texttt{add2}} \texttt{s}(\texttt{s}(\texttt{0})) \prec_{\texttt{add2}} \ldots$$

$$\begin{aligned}&\textbf{fun } \texttt{add2}(x : \texttt{nat}, y : \texttt{nat}) \to \texttt{nat} :=\\ &\quad \textbf{match } x \textbf{ with}\\ &\qquad (\texttt{0} \to y)\\ &\qquad (\texttt{s}(\texttt{0}) \to \texttt{s}(y))\\ &\qquad (\texttt{s}(\texttt{s}(z)) \to \texttt{s}(\texttt{add2}(\texttt{s}(z), y)))\end{aligned}$$

Figure 3.3: The function add2

Therefore, we have the relation $\prec_{\texttt{add2}} \subset \prec_{\texttt{add}}$ as $\prec_{\texttt{add}}$ relates one more pair, namely $\texttt{0} \prec_{\texttt{add}} \texttt{s}(\texttt{0})$. However, if we use the induction formula suggested by add on a formula P containing add2, that is

$$(P[\texttt{0}] \land \forall x.(P[x] \to P[\texttt{s}(x)])) \to \forall x.P[x]$$

the recursive branch of add2 will not match the induction step. This is not a problem here, since performing a second induction once more based on add on the fresh variable of the inductive step conclusion gives the proper case distinction with $\texttt{s}(\texttt{0})$ and $\texttt{s}(\texttt{s}(z))$. In other cases however, it may be a problem.

For that matter, our adaption of containment of well-founded orders to r-description instances and induction schemes is done with the following slightly weaker definition:

Definition 3.6.2 An r-description instance $(\mu, \{\mu_j \mid 1 \leq j \leq m\}, F)$ **contains** another r-description instance $(\nu, \{\nu_j \mid 1 \leq j \leq m\}, G)$ if $dom(\mu) = dom(\nu)$ and:

- for every $s \in dom(\mu)$, there is a variable substitution (i.e. mapping only variables to variables) θ_s s.t. $\nu(s)\theta_s = \mu(s)$
- all ν_j can be mapped to a μ_k s.t. for all $s \in dom(\nu_j)$, either $s \notin dom(\mu_k)$ or $\nu_j(s)\theta_s = \mu_k(s)$
- $F \subseteq G\theta$ where θ is the application of all θ_s for $s \in dom(\mu)$

Given two induction schemes I_1 and I_2, I_1 **contains** I_2 if the r-description instances of I_2 are each contained by those of I_1.

This definition is weaker in the sense that it does not apply to r-description instances like the recursive case of add2 and add. We chose to use this definition not to lose more delicate case distinctions and to only discard trivial cases. For instance, this definition can be used to eliminate duplicate instances of the same induction scheme stemming from different function terms. E.g.:

$$\forall x, y, z.\texttt{add}(x, \texttt{add}(y, z)) = \texttt{add}(\texttt{add}(x, y), z)$$

gives two induction schemes which both map x to a recursive case of a successor and a base case of zero:

$$I_1 = I_2 = \{(\{x \mapsto \mathtt{0}\}, \emptyset, \emptyset), (\{x \mapsto \mathtt{s}(z)\}, \{\{x \mapsto z\}\}, \emptyset)\}$$

Both cases in the r-description can be mapped to themselves by the empty substitution which essentially means that I_1 contains I_2 and vice versa, thus they are equal trivially and either I_1 or I_2 can be discarded. Another example is the induction schemes corresponding to formulas 3.3 and 3.4. This is also trivial with the empty substitution and only the scheme with more hypotheses is kept. All other cases will be dealt with by combining the induction schemes, which we will discuss in Section 3.8. We conclude this section with the following result.

Lemma 3.6.1 Let I_1 and I_2 be two induction schemes. If I_1 contains I_2, then $\prec_{I_2} \subseteq \prec_{I_1}$.

Proof. We have by assumption that each r-description instance $(\nu, \{\nu_j \mid 1 \leq j \leq m\}, G)$ of I_2 is contained by some r-description instance $(\mu, \{\mu_i \mid 1 \leq i \leq n\}, F)$ of I_1. Since $dom(\mu) = dom(\nu)$, every related pair of tuples of $\prec_{I_1}$ and $\prec_{I_2}$ constructed based on Section 3.5 have the same arity. Mapping each subrelation in $\prec_{I_2}$ corresponding to a ν_j-ν pair to a subrelation in $\prec_{I_1}$ corresponding to a μ_i-μ pair with some substitution where also the set of side conditions F is a subset of G after applying the substitution, hence the second entailing the first. Therefore, we get that each such subrelation of $\prec_{I_2}$ is contained in one from $\prec_{I_1}$ and therefore the union of the former subrelations are the subset of the union of the latter, i.e. $\prec_{I_2} \subseteq \prec_{I_1}$. □

3.7 Strengthening induction hypotheses

When an argument of an inductive function term is not used as an induction term during induction, it is essentially not needed for the well-foundedness of the corresponding relation. Consider the following conjecture:

$$\forall x, y.\mathtt{leq}(x, \mathtt{add}(x, y))$$

Here, from the r-descriptions of `leq` and `add` from Equation 3.1 and 3.2, we get that both of `leq`'s and the first of `add`'s argument positions are active. Thus, the only induction term can be x as y has no active occurrences. By the recursive definition of `leq`, we get the well-founded relation $x \prec \mathtt{s}(x)$ for the induction formula:

$$\forall y. \left(\begin{array}{c} \mathtt{leq}(\mathtt{0}, \mathtt{add}(\mathtt{0}, y)) \wedge \\ \forall z. \left(\begin{array}{c} \mathtt{leq}(z, \mathtt{add}(z, y)) \rightarrow \\ \mathtt{leq}(\mathtt{s}(z), \mathtt{add}(\mathtt{s}(z), y)) \end{array} \right) \end{array} \right) \rightarrow \forall x.\mathtt{leq}(x, \mathtt{add}(x, y))$$

This is based on the change in the value of x only, which means we can use any value for y in the induction hypothesis and still get a well-founded relation. This way we strengthen

the induction hypothesis as instead of a fixed y quantified over the whole induction step, we can use a new variable w quantified only over the induction hypothesis:

$$\forall y.\left(\begin{array}{c}\mathtt{leq}(\mathtt{0},\mathtt{add}(\mathtt{0},y))\wedge\\ \forall z.\left(\begin{array}{c}\forall w.\mathtt{leq}(z,\mathtt{add}(z,w))\rightarrow\\ \mathtt{leq}(\mathtt{s}(z),\mathtt{add}(\mathtt{s}(z),y))\end{array}\right)\end{array}\right)\rightarrow\forall x.\mathtt{leq}(x,\mathtt{add}(x,y))$$

The ability to choose any value in place of fixed y in the induction hypothesis increases the chances that the induction will succeed. In light of the previous section, we can say that the well-founded relation corresponding to the induction formula with the strengthened hypothesis relates more pairs. If we extend the relation with the otherwise unused y, instead of relating $(\mathtt{s}(x),y)$ to only (x,y), we relate it to $(x,\mathtt{p}(y))$, $(x,\mathtt{p}(\mathtt{p}(y)))$, ... and $(x,\mathtt{s}(y))$, $(x,\mathtt{s}(\mathtt{s}(y)))$, ... as well.

In terms of r-description instances, we can select any term containing no induction terms to make the induction hypotheses stronger by replacing them with universally quantified variables. Otherwise, we can fix a non-induction term by making it explicit in the substitution:

$$(\{x\mapsto\mathtt{s}(z),y\mapsto w\},\{\{x\mapsto z,y\mapsto w\}\},\emptyset)$$

Here, we substitute a variable w for y in both the main and recursive substitutions to denote that we cannot strengthen the induction hypothesis this way. This r-description instance is contained by the one where we do not make restrictions on y:

$$(\{x\mapsto\mathtt{s}(z)\},\{\{x\mapsto z\}\},\emptyset)$$

Although making such restrictions seems pointless as we weaken the induction hypotheses, we can use such restrictions when combining induction schemes to get a well-founded union (see Section 3.8).

To generate induction formulas with stronger induction hypotheses, we need to modify the rules for generating the induction hypotheses from Section 3.4.1: for each substitution $\mu=\{x_i\mapsto t_i\mid 1\leq i\leq n\}$ in an r-description instance corresponding to an induction hypothesis, given a formula $P[x_1]...[x_n][y_1]...[y_n]$ where $x_1,...,x_n$ are the induction term occurrences and $y_1,...,y_n$ are some occurrences of other terms, we create a formula

$$P_\mu=P[t_1]...[t_n][w_1]...[w_m]$$

where $w_1,...,w_m$ are fresh variables. This ensures that variables in P not used in the well-founded relation of the induction are strengthened.

We can also strengthen the induction hypotheses based on the ordered separation of the induction template. This is done by using a refined definition for r-description instances instead of Definition 3.4.2:

Definition 3.7.1 A function term $\mathtt{f}(s_1, ..., s_n)$ with a tuple of selected terms $(s'_{i_1}, ..., s'_{i_m})$ s.t. for all $1 \leq j \leq m$, $i_j \in I^a_{\mathtt{f}}$ and $s'_{i_j} \trianglelefteq s_{i_j}$, and an r-description

$$(\mathtt{f}(t_1, ..., t_n), \{\mathtt{f}(t_{1j}, ..., t_{nj}) \mid 1 \leq j \leq m\}, F)$$

and ordered separation $I := \{I_k \mid 1 \leq k \leq N\}$ together define a **strengthened r-description instance**

$$(\mu, \{\mu_j \mid 1 \leq j \leq m\}, F')$$

where μ and F' are defined as in the Definition 3.4.2. We create the following helper set for each μ_j containing the necessary active argument positions from I:

$$K_j := \bigcup I_k \text{ s.t. } I_k \in I \text{ and } \forall i \in I_k . t_i \trianglerighteq t_{ij}$$

Then, we define μ_j as:

$$\begin{aligned}\mu_j = \{s \mapsto t \mid \exists J \subseteq K_j . J \neq \emptyset \wedge \forall i \in J . (s = s'_i \wedge t = t_i \cdot mgu(\{t_i \mid i \in J\}))\} \\ \cup \{s \mapsto x \mid \exists J \subseteq (I^a_{\mathtt{f}} \setminus K_j) . J \neq \emptyset \wedge \forall i \in J . s = s'_i \wedge x \text{ is fresh variable}\}\end{aligned}$$

This definition essentially says that using the ordered separation, we can find the indices for induction terms which need to be present as fixed terms in the induction hypotheses in order to get the well-founded order. *All other induction terms in later indices in the ordered separation can be replaced by new variables to strengthen the hypotheses.* The rest of the definition for a strengthened induction scheme is analogous to the original.

Note that the method used in Section 3.5 to obtain a well-founded order works the same on strengthened induction schemes.

Example 3.7.1 Consider the function term $\mathtt{ack}(x, y)$. The function $\mathtt{ack}$ has an ordered separation $(1, 2)$ which can be used to get the strengthened induction scheme. The r-description instances for the base cases stay the same since they contain no recursive cases to strengthen:

$$\{(\{x \mapsto \mathtt{s}(z_0), y \mapsto \mathtt{0}\}, \emptyset, \emptyset), (\{x \mapsto z_1, y \mapsto \mathtt{s(0)}\}, \emptyset, \emptyset)\}$$

For the two induction hypotheses in the recursive case, we generate the indices $k_1 = 1$ and $k_2 = 2$ since between $\mathtt{ack}(\mathtt{s}(z), \mathtt{s}(w))$ and its recursive call $\mathtt{ack}(z, \mathtt{ack}(\mathtt{s}(z), w))$, index 1 is the first changing one and between $\mathtt{ack}(\mathtt{s}(z), \mathtt{s}(w))$ and its recursive call $\mathtt{ack}(\mathtt{s}(z), w)$, the index 1 is not changing so index 2 is the first changing one. The recursive case main substitution also stays unaltered.

We generate the recursive r-description instance with the new variable z_4 strengthening

the first induction hypothesis:

$$(\{x \mapsto \mathtt{s}(z_2), y \mapsto \mathtt{s}(z_3)\}, \{\{x \mapsto z_2, y \mapsto z_4\}, \{x \mapsto \mathtt{s}(z_2), y \mapsto z_3\}, \emptyset\})$$

3.8 Combining induction schemes

When an inductive goal has multiple suggested induction schemes, we have the following possibilities: either all of them has pairwise distinct sets of induction terms or there are some induction schemes with non-empty intersections of induction terms.

In the former case, generating an induction formula from one of the induction schemes will only affect the terms which contain the induction terms of this scheme. Therefore, after simplification, any other of the original schemes may still be applicable on the remaining simplified induction formula. However, there may be induction schemes which succeed without the need for a second round of induction. For that matter, we either need to try all induction schemes in the first place, or only use the most promising ones.

Example 3.8.1 Prove the commutativity of `add`:

$$\forall x, y.\mathtt{add}(x, y) = \mathtt{add}(y, x)$$

If we denote this conjecture with $\forall x, y.P[x][y]$, the two suggested induction schemes are:

$$\forall y.((P[\mathtt{0}][y] \land \forall x_0.(P[x_0][y] \to P[\mathtt{s}(x_0)][y])) \to \forall x.P[x][y])$$

and

$$\forall x.((P[x][\mathtt{0}] \land \forall y_0.(P[x][y_0] \to P[x][\mathtt{s}(y_0)])) \to \forall y.P[x][y])$$

The order of these induction schemes are interchangeable, one of them result in two subgoals $y = \mathtt{add}(y, \mathtt{0})$ and $\mathtt{s}(\mathtt{add}(x_0, y)) = \mathtt{add}(y, \mathtt{s}(x_0))$, on both of them we can apply the second scheme, likewise if we use the second scheme first.

In case of intersecting induction terms, consider the following example, formalizing transitivity of `leq`:

$$\forall x, y, z.(\mathtt{leq}(x, y) \land \mathtt{leq}(y, z)) \to \mathtt{leq}(x, z)$$

The first `leq` occurrence suggests an induction over x and y, the second on y and z and the third on x and z. If we induct on only two of these variables, say x and y, the induction would fail in the inductive step, as we could simplify the first `leq` occurrence but not the two other and hence the induction hypothesis would only partially match. If we induct once more with another induction scheme, the problem would be the same as we could not get rid of all terms at once to match the induction hypothesis.

One solution is to strenghten the induction hypothesis of one of the suggested formulas, after which we can match up the previously unmatched successor terms to apply the

hypothesis. Another solution, suggested by the heuristics of ACL2 [BM79, MW13] is to combine the three schemes, resulting in a successful induction formula. Let us denote the transitivity conjecture as $Q[x][y][z]$:

$$(Q[\mathtt{0}][\mathtt{0}][\mathtt{0}] \wedge \forall x_0, y_0, z_0.(Q[x_0][y_0][z_0] \rightarrow Q[\mathtt{s}(x_0)][\mathtt{s}(y_0)][\mathtt{s}(z_0)])) \rightarrow \forall x, y, z.Q[x][y][z]$$

As described in [Wal92], this method is very successful in practice despite the fact that it uses the *intersection* of induction schemes instead of the *union* which is what properly contains all original induction schemes. We can use the *separated union* to merge induction schemes with identical sets of induction terms which is equivalent to the union as described in [Wal93]:

Definition 3.8.1 The **separated union for two induction schemes** $I_1 = \{(\mathtt{id}, \{\mu_{1ij} \mid 1 \leq j \leq n_i\}, F_{1i}) \mid 1 \leq i \leq n\}$ and $I_2 = \{(\mathtt{id}, \{\mu_{2kl} \mid 1 \leq l \leq m_k\}, F_{2k}) \mid 1 \leq k \leq m\}$ is denoted with $I_1 \otimes I_2$ and is the smallest set s.t. for any $1 \leq i \leq n$ and $1 \leq k \leq m$:

- $(\mathtt{id}, \{\mu_{1ij} \mid 1 \leq j \leq n_i\}, F_{1i} \cup \overline{F_2}) \in I_1 \otimes I_2$
- $(\mathtt{id}, \{\mu_{2kl} \mid 1 \leq l \leq m_k\}, F_{2k} \cup \overline{F_1}) \in I_1 \otimes I_2$
- $(\mathtt{id}, \{\mu_{1ij} \mid 1 \leq j \leq n_i\} \cup \{\mu_{2kl} \mid 1 \leq l \leq m_k\}, F_{1i} \cup F_{2k}) \in I_1 \otimes I_2$

where $\overline{F_1} = \{\neg(\wedge_{f \in F_{1i}} f) \mid 1 \leq i \leq n\}$ and $\overline{F_2} = \{\neg(\wedge_{f \in F_{2k}} f) \mid 1 \leq k \leq m\}$

Notice that this definition applies only to destructor-based definitions, where the inductive step substitution is $\mathtt{id}$, that is, we substitute itself for every induction term. This definition creates all combinations or the "cross-product" of r-description instances of the two induction schemes. The first two sets of r-description instances are the ones which correspond to original cases but exclude all other cases from the other scheme. The last one corresponds to the pairwise intersections of the two schemes.

We now adapt this to constructor-based definitions without any side conditions with the help of *unifiability*. This will help to introduce less conditions in the final induction formula which helps to get smaller normal forms in a prover. A constructor-based r-description instance $(\mu, \{\mu_j \mid 1 \leq j \leq n\}, \emptyset)$ can be reformulated to have the conditions on the step formula given by the substitution μ as $\bigwedge_{s \in dom(\mu)} s = \mu(s)$. Then, two such r-description instances $(\mu, \{\mu_j \mid 1 \leq j \leq n\}, \emptyset)$ and $(\nu, \{\nu_j \mid 1 \leq j \leq m\}, \emptyset)$ apply at the same time if and only if:

$$\bigwedge_{s \in dom(\mu)} s = \mu(s) \wedge \bigwedge_{t \in dom(\nu)} t = \nu(t)$$

If the two merged r-description instances both contain equations on the same term t, we can reduce the question of whether these equations hold to whether the terms t equal to in these equations are unifiable. The unification of these then gives the main

substitution of the resulting merged r-description instance. Also, we need to apply the same unification on the recursive substitutions as well. In the end, no new conditions will be introduced.

Before we can achieve this, we extend most general unifiers to substitutions to obtain shorter definitions:

Definition 3.8.2 A **most general unifier for two substitutions** μ and ν (denoted by $mgu(\mu, \nu)$) is a mapping of elements $s \in dom(\mu) \cup dom(\nu)$ to most general unifiers of terms s.t.:

$$mgu(\mu, \nu)(s) := \begin{cases} mgu(\mu(s), \nu(s)), & \text{if } s \in dom(\mu) \cap dom(\nu) \\ \emptyset, & \text{otherwise} \end{cases}$$

The **application of a most general unifier of two substitution** μ **and** μ' **to a substitution** ν is the following substitution:

$$\nu \cdot mgu(\mu, \mu') := \{s \mapsto \nu(s) \cdot mgu(\mu, \mu')(s) \mid s \in dom(\nu)\}$$

Then, the intersection of two r-description instances is defined as follows:

Definition 3.8.3 The **intersection** of two r-description instances $R_1 := (\mu, \{\mu_j \mid 1 \leq j \leq n\}, \emptyset)$ and $R_2 := (\nu, \{\nu_j \mid 1 \leq j \leq m\}, \emptyset)$ is denoted by $R_1 \cap R_2$ and exists if for each element in $s \in dom(\mu) \cap dom(\nu)$, $\mu(s)$ and $\nu(s)$ are unifiable. The intersection is then:

$$R_1 \cap R_2 := (\mu \cdot \sigma \cup \nu \cdot \sigma, \{\mu_j \cdot \sigma \mid 1 \leq j \leq n\} \cup \{\nu_j \cdot \sigma \mid 1 \leq j \leq m\}, \emptyset)$$

where $\sigma := mgu(\mu, \nu)$

Apart from the pairwise intersections between the r-description instances, to get the full union between two induction schemes, we must also add r-description instances on their own – that is, by excluding all r-description instances of the other scheme. Given a second induction scheme with main substitutions $\{\nu_j \mid 1 \leq j \leq m\}$ from its r-description instances, instead of adding the conditions given by these positively, we negate these conditions, resulting in disjunctions for each such ν_j:

$$\bigwedge_{s \in dom(\mu)} s = \mu(s) \wedge \bigwedge_{1 \leq j \leq m} \bigvee_{t \in dom(\nu_j)} t \neq \nu(t)$$

This formula can be placed as a new condition in the resulting r-description instance, which in turn can be simplified by a prover in the final induction formula. However, as already mentioned we would like to burden any inference engine the least amount

with additional side conditions. In this case it makes even more sense: the negation and applying distributivity often result in very large formulas.

To avoid this, we define instead the operation of excluding r-description instances from one another such that the conditions are "folded back" into the substitutions and hence, no additional condition is generated. First, we need to know what constructors each term algebra for the arguments has, so we can generate the combinations of terms that are still valid after excluding the other r-description instances. We first define how we can exclude one constructor term from a set of constructor terms:

Definition 3.8.4 Given an inductive type τ, a **set of available terms for** τ is

$$\{c_1(\overline{x_1}), ..., c_n(\overline{x_n})\}$$

where c_i are the constructors of τ of arity m_i and $\overline{x_i}$ are tuples of fresh variables of arity m_i of the appropiate sorts.

An inductive type τ has an infinite number of available term sets, given that each contains different variables.

Definition 3.8.5 Given a set of mutually exclusive (i.e. non-unifiable) **available terms** $\{t_1, ..., t_n\}$ containing only constructor terms or one variable for an inductive type τ and a constructor term s of type τ, the **exclusion of** s **from** $\{t_1, ..., t_n\}$ is the smallest set T, s.t. for each $1 \leq i \leq n$:

- If t_i and s are not unifiable, then $t_i \in T$.
- Otherwise, if t_i does not equal to $t_i \cdot mgu(t_i, s)$ up to variable renaming (i.e. s is an instance of t_i), we take the set of positions P where for each $p \in P$, $t_i|_p$ is a variable and $s|_p$ is not.

  For each $p \in P$, we exclude $s|_p$ of type τ' recursively from a newly generated set of available terms for τ'.

  The resulting terms $\{t'_1, ..., t'_k\}$ give rise to a set of substitutions $\nu_p = \{\{t_i|_p \mapsto t'_1\}, ..., \{t_i|_p \mapsto t'_k\}\}$. Then, $\{t_i\nu \mid \nu \in \bigcup_{p \in P} \nu_p\} \subseteq T$.

Let us now illustrate this with an example:

Example 3.8.2 We exclude the term $\texttt{node}(\texttt{node}(x_0, x_1, x_2), x_3, \texttt{leaf})$ from a set of available terms $\{\texttt{leaf}, \texttt{node}(y_0, y_1, y_2)\}$ resulting in the set T as follows:

- $\texttt{leaf}$ and $\texttt{node}(\texttt{node}(x_0, x_1, x_2), x_3, \texttt{leaf})$ are not unifiable, hence $\texttt{leaf} \in T$.

- $t := \texttt{node}(y_0, y_1, y_2)$ and $s := \texttt{node}(\texttt{node}(x_0, x_1, x_2), x_3, \texttt{leaf})$ are unifiable with mgu $\sigma = \{y_0 \mapsto \texttt{node}(x_0, x_1, x_2), y_2 \mapsto \texttt{leaf}\}$ and $t\sigma$ is not equal to t up to variable renaming.

  For the set of positions $P = \{1, 3\}$ we generate sets of constructor terms $\{\texttt{leaf}, \texttt{node}(u_0, u_1, u_2)\}$ and $\{\texttt{leaf}, \texttt{node}(w_0, w_1, w_2)\}$. We exclude $s|_1$, i.e. $\texttt{node}(x_0, x_1, x_2)$ from the first. The result contains only `leaf` as it is not unifiable with $s|_1$ and $\texttt{node}(u_0, u_1, u_2)$ is unifiable but the application of the unification is equal to $s|_1$ up to variable renaming. We exclude $s|_3$ similarly from the second constructor term set, which results in $\{\texttt{node}(w_0, w_1, w_2)\}$.

  The sets of substitutions are then $\{\{t|_1 \mapsto \texttt{leaf}\}\}$ and $\{\{t|_3 \mapsto \texttt{node}(w_0, w_1, w_2)\}\}$. Taking the union of these and applying each to t, we get the set

$$\{\texttt{node}(\texttt{leaf}, y_1, y_2), \texttt{node}(y_0, y_1, \texttt{node}(w_0, w_1, w_2))\}$$

So the final result is:

$$\{\texttt{leaf}, \texttt{node}(\texttt{leaf}, y_1, y_2), \texttt{node}(y_0, y_1, \texttt{node}(w_0, w_1, w_2))\}$$

Before defining the exclusion of r-description instances from an r-description instance R, we show how we maintain a set of mappings from the induction terms to their available terms in the main substitution of R.

Definition 3.8.6 An **available term mapping** T is a mapping from terms to sets of terms. The **available term mapping for an r-description instance** $R := (\mu, \{\mu_j \mid 1 \leq j \leq n\}, F)$ is defined as a mapping:

$$T_R := \{s \mapsto \{\mu(s)\} \mid s \in dom(\mu)\}$$

The **available term mapping for a set of inductive terms** $\mathcal{D}$ is:

$$T_{\mathcal{D}} := \{s \mapsto \{x\} \mid s \in \mathcal{D} \text{ and } x \text{ is a fresh variable}\}$$

The **normalization for an available term mapping** T is a set of mappings $N(T)$ from terms to terms defined as:

$$N(T) := \bigtimes_{s \in dom(T)} \{\{s \mapsto t\} \mid t \in T(s)\}$$

Each available term mapping is a different case for an r-description instance. The normalization is the "cross-product" of this case, collecting all available substitution combinations.

Definition 3.8.7 Given an r-description $R = (\mu, \{\mu_j \mid 1 \leq j \leq n\}, \emptyset)$ and a set of r-description instances $\mathcal{R} := \{R_i := (\nu_i, N, \emptyset) \mid 1 \leq i \leq m\}$, the **exclusion of the set of r-description instances $\mathcal{R}$ from r-description instance** R is a set of r-description instances denoted by $R \setminus \mathcal{R}$.

Let $\mathcal{D} := dom(\mu) \cup \bigcup_{1 \leq i \leq m} dom(\nu_i)$. We define $k+1$ sets of available term mappings $\mathcal{T}_0, ..., \mathcal{T}_k$ iteratively:

$$\mathcal{T}_0 := \{T_R \cup T_{\mathcal{D} \setminus dom(\mu)}\}$$

For each $0 \leq i < k$, $s \in \mathcal{D}$ and $T \in \mathcal{T}_i$, if $s \in dom(\nu_i)$ we define $T' \in \mathcal{T}_{i+1}$ on elements $t \in dom(T)$ as:

$$T'(t) := \begin{cases} \text{exclusion of } \nu_i(s) \text{ from } T(s), & \text{if } s = t \\ T(t), & \text{otherwise} \end{cases}$$

Then the exclusion of the set $\mathcal{R}$ from R is:

$$R \setminus \mathcal{R} := \{(\mu\sigma, \{\mu_j\sigma \mid 1 \leq j \leq n\}, \emptyset) \mid \sigma := mgu(\mu, \nu), \nu \neq \emptyset, \nu \in N(T), T \in \mathcal{T}_k\}$$

For a combined induction scheme for I_1 and I_2, we only create the intersections and exclusions between the recursive cases, i.e. between elements of I_1^{rec} and I_2^{rec}. After computing the intersections and exclusions, we can simply check what combinations of argument tuples are not covered by this resulting set. Therefore the base cases are generated by excluding the recursive cases of the combined induction scheme from all available constructor term combinations for the induction terms:

Definition 3.8.8 The **set of base cases for a set of r-description instances** $\mathcal{R} := \{R_1, ..., R_n\}$ without any conditions and with main substitutions $\mu_1, ..., \mu_n$ and $\mathcal{D} := \cup_{1 \leq i \leq n} dom(\mu_i)$, is denoted by $B(\mathcal{R})$ and defined as follows.

We define $n+1$ sets of available term mappings $\mathcal{T}_0, ..., \mathcal{T}_n$ iteratively:

$$\mathcal{T}_0 := \{T_{\mathcal{D}}\}$$

For each $0 \leq i < n$, $s \in \mathcal{D}$ and $T \in \mathcal{T}_i$, if $s \in dom(\mu_i)$ we define $T' \in \mathcal{T}_{i+1}$ on elements $t \in dom(T)$ as:

$$T'(t) := \begin{cases} \text{exclusion of } \mu_i(s) \text{ from } T(s), & \text{if } s = t \\ T(t), & \text{otherwise} \end{cases}$$

The set of base cases is:

$$B(\mathcal{R}) := \{(\nu, \emptyset, \emptyset) \mid \nu \neq \emptyset, \nu \in N(T), T \in \mathcal{T}_n\}$$

Now we can define what the union of two induction schemes is:

Definition 3.8.9 The **union of two induction schemes** I_1 and I_2 is denoted by $I_{1\cup 2}$ and consists of the following recursive r-description instances:

$$I^{rec}_{1\cup 2} := \{R_j \setminus I^{rec}_2, Q_l \setminus I^{rec}_1, R_j \cap Q_l \mid R_j \in I^{rec}_1, Q_l \in I^{rec}_2\}$$

The union itself is then:

$$I_{1\cup 2} := I^{rec}_{1\cup 2} \cup B(I^{rec}_{1\cup 2})$$

This means that for two induction schemes with n and m r-description instances, the exclusions give at most $n+m$ new r-description instances while the pairwise intersections give at most $n \cdot m$ new ones. One may wonder why is it not necessary to include every combination of intersections and exclusions. That is based on the well-definedness of the induction schemes (and therefore the original functions). Since each r-description in the original induction template only matches one argument tuple, the resulting r-description instances must be mutually exclusive and therefore any other combination would result in either duplicates or invalid cases.

Let us try to create the union in a simple example:

Example 3.8.3 Given two r-description instances:

$$R_1 = (\{x \mapsto \mathtt{s}(\mathtt{s}(z))\}, \{\{x \mapsto z\}\}, \emptyset)$$

$$R_2 = (\{x \mapsto \mathtt{s}(w)\}, \{\{x \mapsto w\}\}, \emptyset)$$

We start by creating the exclusions of the two r-description instances from one another. The set of initial available term mappings for R_1 is:

$$\mathcal{T}_0 = \{\{x \mapsto \{\mathtt{s}(\mathtt{s}(z))\}\}\}$$

We exclude only R_2, so the final set of mappings is:

$$\mathcal{T}_1 = \{\emptyset\}$$

The reason for this is that the exclusion of $\mathtt{s}(w)$ from $\{\mathtt{s}(\mathtt{s}(z))\}$ results in one unification where we do not have any positions where $\mathtt{s}(\mathtt{s}(z))$ contains a variable but $\mathtt{s}(w)$ does not. Therefore, this case is empty as the normalization of $\mathcal{T}_1$ contains no substitution.

In the exclusion of $\{R_1\}$ from R_2, we have the initial set:

$$\mathcal{T}_0 = \{\{x \mapsto \{\mathtt{s}(w)\}\}\}$$

We exclude $\mathtt{s}(\mathtt{s}(z))$ from $\{\mathtt{s}(w)\}$. The position where we have w in one and $\mathtt{s}(z)$ in the other matches the second condition of the definition, so we create available terms $\{\mathtt{0}, \mathtt{s}(u)\}$ and exclude $\mathtt{s}(z)$ from these. Here, $\mathtt{0}$ is not unifiable, but $\mathtt{s}(u)$ is with $\mathtt{s}(z)$

and there is no position satisfying the second condition of the definition. Therefore, the only available term is $\mathtt{0}$ in the subresult. We substitute this in for w and we get:

$$\mathcal{T}_1 = \{\{x \mapsto \{\mathtt{s}(\mathtt{0})\}\}\}$$

We normalize this to get:

$$N(\mathcal{T}_1) = \{\{x \mapsto \mathtt{s}(\mathtt{0})\}\}$$

We now apply the only substitution in this on the original r-description instance to get:

$$R_2 \setminus \{R_1\} = \{(\{x \mapsto \mathtt{s}(\mathtt{0})\}, \{\{x \mapsto \mathtt{0}\}\}, \emptyset)\}$$

Next, we create the intersection. For this, we unify $\mathtt{s}(\mathtt{s}(z))$ and $\mathtt{s}(w)$ which results in the mgu $\{w \mapsto \mathtt{s}(z)\}$. We apply this on the original substitutions of R_1 and R_2 and take their union:

$$R_1 \cap R_2 := (\{x \mapsto \mathtt{s}(\mathtt{s}(z))\}, \{\{x \mapsto \mathtt{s}(z)\}, \{x \mapsto z\}\}, \emptyset)$$

Lastly, we need to generate the base cases. For this, we generate an available term mapping with all `nat` constructors:

$$\mathcal{T}_0 = \{\{x \mapsto \{\mathtt{0}, \mathtt{s}(x_0)\}\}\}$$

We need to first exclude all elements of $R_2 \setminus \{R_1\}$, that is, $\mathtt{s}(\mathtt{s}(z))$. This gives the first iteration:

$$\mathcal{T}_1 = \{\{x \mapsto \{\mathtt{0}, \mathtt{s}(\mathtt{0})\}\}\}$$

Next, we exclude the intersection $R_1 \cap R_2$, that is $\mathtt{s}(\mathtt{0})$ from $\mathcal{T}_1$. This eliminates $\mathtt{s}(\mathtt{0})$ from the possibilities, so we get:

$$\mathcal{T}_2 = \{\{x \mapsto \{\mathtt{0}\}\}\}$$

Hence, our base case set is:

$$B(R_1 \setminus \{R_2\} \cup \{R_2 \cap R_1\}) := \{(\{x \mapsto \mathtt{0}\}, \emptyset, \emptyset)\}$$

Finally, the union is:

$$\left\{ \begin{array}{lll} (\{x \mapsto \mathtt{0}\}, & \emptyset, & \emptyset), \\ (\{x \mapsto \mathtt{s}(\mathtt{0})\}, & \{\{x \mapsto \mathtt{0}\}\}, & \emptyset), \\ (\{x \mapsto \mathtt{s}(\mathtt{s}(z))\}, & \{\{x \mapsto \mathtt{s}(z)\}, \{x \mapsto z\}\}, & \emptyset) \end{array} \right\}$$

In the previous example, the exclusions seemed a bit overcomplicated since it only contained one induction term and the resulting sets had only one substitution each. To illustrate how multiple induction terms can give more r-description instances in the exclusion steps, let us look at a more complicated example with two induction terms:

Example 3.8.4 We are given two r-description instances:

$$R_1 = \Big(\{x \mapsto \texttt{node}(z_0, z_1, z_2)\}, \{\{x \mapsto z_0\}\}, \emptyset\Big)$$

$$R_2 = \left(\left\{\begin{matrix} x \mapsto \texttt{node}(\texttt{node}(w_0, w_1, w_2), w_3, w_4), \\ y \mapsto \texttt{s}(v) \end{matrix}\right\}, \left\{\left\{\begin{matrix} x \mapsto w_4, \\ y \mapsto v \end{matrix}\right\}\right\}, \emptyset\right)$$

The set of available term mappings for R_1 is the following with y_0 fresh variable:

$$\mathcal{T}_0 = \{\{x \mapsto \{\texttt{node}(z_0, z_1, z_2)\}, y \mapsto \{y_0\}\}\}$$

We first exclude R_2 from R_1:

- First exclude $\texttt{node}(\texttt{node}(w_0, w_1, w_2), w_3, w_4)$ from $\{\texttt{node}(z_0, z_1, z_2)\}$. After unification of the only available term with the term we want to exclude, we find that there is a position containing a variable z_0 which needs a case distinction. We generate for this position the available terms $\{\texttt{leaf}, \texttt{node}(z_3, z_4, z_5)\}$. `leaf` is not unifiable with $\texttt{node}(w_0, w_1, w_2)$ therefore it is in the result, but $\texttt{node}(z_3, z_4, z_5)$ is unifiable and there is no position we can make a case-distinction on. So only `leaf` is the term we can substitute in $\texttt{node}(z_0, z_1, z_2)$ for z_0. This results in:

$$\{\{x \mapsto \{\texttt{node}(\texttt{leaf}, z_1, z_2)\}, y \mapsto \{y_0\}\}\}$$

- Next exclude $\texttt{s}(v)$ from $\{y_0\}$. This expands the set first to $\{\texttt{0}, \texttt{s}(y_1)\}$. $\texttt{s}(v)$ and $\texttt{s}(y_1)$ are unifiable, and position 1 in $\texttt{s}(y_1)$ cannot be further expanded. Therefore, our resulting set is:

$$\{\{x \mapsto \{\texttt{node}(z_0, z_1, z_2)\}, y \mapsto \{\texttt{0}\}\}\}$$

Therefore, we get:

$$\mathcal{T}_1 = \left\{\begin{matrix} \{x \mapsto \{\texttt{node}(\texttt{leaf}, z_1, z_2)\}, & y \mapsto \{y_0\}\}, \\ \{x \mapsto \{\texttt{node}(z_0, z_1, z_2)\}, & y \mapsto \{\texttt{0}\}\} \end{matrix}\right\}$$

We normalize the two available term mappings and apply the resulting substitutions to get the two r-description instances of $R_1 \setminus \{R_2\}$:

$$R_1 \setminus \{R_2\} = \left\{\begin{matrix} (\{x \mapsto \texttt{node}(\texttt{leaf}, z_1, z_2), y \mapsto y_0\}, & \{\{x \mapsto \texttt{leaf}\}\}, & \emptyset), \\ (\{x \mapsto \texttt{node}(z_0, z_1, z_2), y \mapsto \texttt{0}\}, & \{\{x \mapsto z_0\}\}, & \emptyset) \end{matrix}\right\} \tag{3.5}$$

Next we do the exclusion of $\{R_1\}$ from R_2. In R_2 the set of available term mappings is:

$$\mathcal{T}_0 := \{\{x \mapsto \{\texttt{node}(\texttt{node}(w_0, w_1, w_2), w_3, w_4)\}, y \mapsto \{\texttt{s}(v)\}\}\}$$

Then we do the following steps:

- We exclude $\texttt{node}(z_0, z_1, z_2)$ from $\{\texttt{node}(\texttt{node}(w_0, w_1, w_2), w_3, w_4)\}$. Here the result is the empty set of substitutions as after the unification we do not have any positions in $\texttt{node}(\texttt{node}(w_0, w_1, w_2), w_3, w_4)$ that contains a variable while in $\texttt{node}(z_0, z_1, z_2)$ it does not. Hence the result is the following:

$$\{\{x \mapsto \emptyset, y \mapsto \{\texttt{s}(v)\}\}\}$$

- Since y is not in the domain of R_1's main substitution, we create no new available term mapping. This makes sense as if we think about the missing y in R_2 as an implicit variable, the exclusion would also result in the empty set.

After normalization of the only available term mapping, we get empty substitutions – this is because it contains an empty available term set. As empty substitutions are not used in the result, the result is empty:

$$R_2 \setminus \{R_1\} = \emptyset$$

We now create the intersection of R_1 and R_2 by unifying $\texttt{node}(z_0, z_1, z_2)$ with $\texttt{node}(\texttt{node}(w_0, w_1, w_2), w_3, w_4)$ resulting in the most general unifiers:

$$\left\{\begin{array}{c} x \mapsto \{z_0 \mapsto \texttt{node}(w_0, w_1, w_2), z_1 \mapsto w_3, z_2 \mapsto w_4\}, \\ y \mapsto \emptyset \end{array}\right\}$$

We then apply these substitutions on the main and recursive call substitutions and the conditions of both R_1 and R_2 to get the intersection:

$$R_1 \cap R_2 = \left(\left\{\begin{array}{c} x \mapsto \texttt{node}(\texttt{node}(w_0, w_1, w_2), w_3, w_4), \\ y \mapsto \texttt{s}(v) \end{array}\right\}, \left\{\left\{\begin{array}{c} x \mapsto \texttt{node}(w_0, w_1, w_2), \\ y \mapsto \texttt{s}(v) \\ \{x \mapsto w_4\} \end{array}\right\}, \right\}, \emptyset\right)$$

In the end, we generate the base cases from the three r-description instances we got from the intersection and exclusions. The initial available term mapping for this is:

$$\mathcal{T}_0 = \{\{x \mapsto \{x_0\}, y \mapsto \{y_1\}\}\}$$

Given the first r-description instance of Equation 3.5, we exclude $\texttt{node}(\texttt{leaf}, z_1, z_2)$ from $\{x_0\}$ and y_0 from $\{y_1\}$. The former results in the expansion of x_0 as $\{\texttt{leaf}, \texttt{node}(v_0, v_1, v_2)\}$ from which we exclude $\texttt{node}(\texttt{leaf}, z_1, z_2)$. The latter results in the empty set. We get:

$$\mathcal{T}_1 = \left\{ \begin{matrix} \{x \mapsto \{\texttt{leaf}, \texttt{node}(\texttt{node}(x_1, x_2, x_3), v_1, v_2)\}, y \mapsto \{y_1\}\}, \\ \{x \mapsto \{x_0\}, y \mapsto \emptyset\} \end{matrix} \right\}$$

We then exclude the second r-description instance of Equation 3.5 from $\mathcal{T}_1$. First we exclude $\texttt{node}(z_1, z_2, z_3)$ which leaves only `leaf` in the corresponding sets. We get the partial result:

$$\left\{ \begin{matrix} \{x \mapsto \{\texttt{leaf}\}, & y \mapsto \{y_1\}\}, \\ \{x \mapsto \{\texttt{leaf}\}, & y \mapsto \emptyset\} \end{matrix} \right\}$$

After this, we exclude `0` resulting in the other part:

$$\left\{ \begin{matrix} \{x \mapsto \{\texttt{leaf}, \texttt{node}(\texttt{node}(x_1, x_2, x_3), v_1, v_2)\}, & y \mapsto \{\texttt{s}(y_2)\}\} \\ \{x \mapsto \{x_0\}, & y \mapsto \emptyset\} \end{matrix} \right\}$$

Altogether, the result of this step is:

$$\mathcal{T}_2 = \left\{ \begin{matrix} \{x \mapsto \{\texttt{leaf}\}, & y \mapsto \{y_1\}\}, \\ \{x \mapsto \{\texttt{leaf}\}, & y \mapsto \emptyset\}, \\ \{x \mapsto \{\texttt{leaf}, \texttt{node}(\texttt{node}(x_1, x_2, x_3), v_1, v_2)\}, & y \mapsto \{\texttt{s}(y_2)\}\} \\ \{x \mapsto \{x_0\}, & y \mapsto \emptyset\} \end{matrix} \right\}$$

Finally, we exclude the intersection from $\mathcal{T}_2$, excluding $\texttt{node}(\texttt{node}(w_0, w_1, w_2), w_3, w_4)$ and $\texttt{s}(v)$. The first gives:

$$\left\{ \begin{matrix} \{x \mapsto \{\texttt{leaf}\}, & y \mapsto \{y_1\}\}, \\ \{x \mapsto \{\texttt{leaf}\}, & y \mapsto \emptyset\}, \\ \{x \mapsto \{\texttt{leaf}\}, & y \mapsto \{\texttt{s}(y_2)\}\} \\ \{x \mapsto \{\texttt{leaf}, \texttt{node}(\texttt{leaf}, x_1, x_2)\}, & y \mapsto \emptyset\} \end{matrix} \right\}$$

The second is:

$$\left\{ \begin{matrix} \{x \mapsto \{\texttt{leaf}\}, & y \mapsto \{\texttt{0}\}\}, \\ \{x \mapsto \{\texttt{leaf}\}, & y \mapsto \emptyset\}, \\ \{x \mapsto \{\texttt{leaf}, \texttt{node}(\texttt{node}(x_1, x_2, x_3), v_1, v_2)\}, & y \mapsto \emptyset\} \\ \{x \mapsto \{x_0\}, & y \mapsto \emptyset\} \end{matrix} \right\}$$

The sets that contain empty available term sets do not give any cross-product. Putting together the two sets to get $\mathcal{T}_3$ and normalizing them gives:

$$\left\{ \begin{array}{ll} \{x \mapsto \texttt{leaf}, & y \mapsto y_1\}, \\ \{x \mapsto \texttt{leaf}, & y \mapsto \texttt{s}(y_2)\} \\ \{x \mapsto \texttt{leaf}, & y \mapsto \texttt{0}\} \end{array} \right\}$$

In the end we get the following set of base cases:

$$B((R_1 \setminus \{R_2\}) \cup \{R_1 \cap R_2\}) = \left\{ \begin{array}{c} (\{x \mapsto x_0, y \mapsto \texttt{0}\}, \emptyset, \emptyset), \\ (\{x \mapsto \texttt{leaf}, y \mapsto \texttt{0}\}, \emptyset, \emptyset), \\ (\{x \mapsto \texttt{leaf}, y \mapsto \texttt{s}(v_3)\}, \emptyset, \emptyset) \end{array} \right\}$$

This contains duplicates, so we can discard either the first or the last two to get a well-defined case distinction.

Returning to the example from Section 3.6 with `add` and `add2`, we can merge the two to get a scheme which essentially matches all cases of `add2` yet has the same well-founded order as `add`:

$$\left\{ \begin{array}{lll} (\{x \mapsto \texttt{0}\}, & \emptyset, & \emptyset), \\ (\{x \mapsto \texttt{s(0)}\}, & \{\{x \mapsto \texttt{0}\}\}, & \emptyset), \\ (\{x \mapsto \texttt{s(s}(z))\}, & \{\{x \mapsto \texttt{s}(z)\}\}, & \emptyset) \end{array} \right\}$$

Despite the fact that the union of well-founded induction schemes is a better candidate for a successful induction since the order corresponding to it contains the composing well-founded orders, this order is not necessarily well-founded. For instance the relations $\prec_1$ and $\prec_2$ defined by $(\texttt{s}(x), y) \prec_1 (x, \texttt{s}(y))$ and $(x, \texttt{s}(y)) \prec_2 (\texttt{s}(x), y)$, respectively, are clearly well-founded taken separately, since at least one tuple element is getting smaller. Taking every possible combination of the `nat` term algebra into account, their union $\prec_{1\cup2}$ contains the relations:

- $(\texttt{s}(\texttt{0}), y) \prec_{1\cup2} (\texttt{0}, \texttt{s}(y))$
- $(x, \texttt{s}(\texttt{0})) \prec_{1\cup2} (\texttt{s}(x), \texttt{0})$
- $(\texttt{s}(\texttt{s}(x)), y) \prec_{1\cup2} (\texttt{s}(x), \texttt{s}(y))$
- $(x, \texttt{s}(\texttt{s}(y))) \prec_{1\cup2} (\texttt{s}(x), \texttt{s}(y))$

A counterexample for the well-foundedness of $\prec_{1\cup2}$ is then the infinite chain:

$$\ldots \prec_{1\cup2} (\texttt{s}(\texttt{0}), \texttt{0}) \prec_{1\cup2} (\texttt{0}, \texttt{s}(\texttt{0})) \prec_{1\cup2} (\texttt{s}(\texttt{0}), \texttt{0}) \prec_{1\cup2} \ldots$$

So before using the union and discarding the composing schemes, we have to check it for well-foundedness. As all induction templates we create admit a lexicographic order with the subterm suborder already discussed in Section 3.3, we now adapt this to induction schemes to be able to check well-foundedness of the union.

Definition 3.8.10 A set of r-description instances $\mathcal{R} := \{(\mu_i, M_i, F_i) \mid 1 \leq i \leq n\}$ is **well-founded** if (1) for all $1 \leq i \leq n$, $M_i = \emptyset$, or (2) there is a $s \in \cup_{1 \leq i \leq n} dom(\mu_i)$ such that for all $1 \leq i \leq n$ and all $\mu \in M_i$, we have $s \in dom(\mu) \cap dom(\mu_i)$ and $\mu(s) \trianglelefteq \mu_i(s)$, moreover

$$\{(\nu_i, N_i, F_i) \mid 1 \leq i \leq n\}$$

is well-founded where

$$\nu_i := \{t \mapsto \mu_i(t) \mid t \in dom(\mu_i) \setminus \{s\}\}$$

and

$$N_i := \{\nu \mid \nu := \{t \mapsto \mu(t) \mid t \in dom(\mu) \setminus \{s\}, \mu_i(s) = \mu(s) \text{ and } \mu \in M_i\}\}$$

In practice, this essentially a check we can perform after the union has been generated. Let us first look at an example where the union is not well-founded.

Example 3.8.5 Let us look at the commutativity of `add` once again from Example 3.8.1. It gives the two induction schemes with distinct induction term sets $\{x\}$ and $\{y\}$:

$$I_1 := \left\{ \begin{array}{ccc} (\{x \mapsto \mathtt{s}(x_0)\}, & \{\{x \mapsto x_0\}\}, & \emptyset), \\ (\{x \mapsto \mathtt{0}\}, & \emptyset, & \emptyset) \end{array} \right\}$$

$$I_2 := \left\{ \begin{array}{ccc} (\{y \mapsto \mathtt{s}(y_0)\}, & \{\{y \mapsto y_0\}\}, & \emptyset), \\ (\{y \mapsto \mathtt{0}\}, & \emptyset, & \emptyset) \end{array} \right\}$$

Their union is the induction scheme:

$$I_{1\cup 2} := \left\{ \begin{array}{ccc} (\{x \mapsto \mathtt{s}(x_0), y \mapsto \mathtt{0}\}, & \{\{x \mapsto x_0\}\}, & \emptyset), \\ (\{x \mapsto \mathtt{0}, y \mapsto \mathtt{s}(y_0)\}, & \{\{y \mapsto y_0\}\}, & \emptyset), \\ (\{x \mapsto \mathtt{s}(x_0), y \mapsto \mathtt{s}(y_0)\}, & \{\{x \mapsto x_0\}, \{y \mapsto y_0\}\}, & \emptyset), \\ (\{x \mapsto \mathtt{0}, y \mapsto \mathtt{0}\}, & \emptyset, & \emptyset) \end{array} \right\}$$

To check well-foundedness, we can apply the definitions by selecting either x or y, removing them from the substitutions and removing the relations where their mapped terms are related via the strict subterm relation. Notice that the first two recursive substitutions do not contain these values at all so the definition does not apply, as in fact, the union is not well-founded.

This can be mitigated by fixing either x in the recursive case of I_2 or y in the recursive case of I_1, weakening the induction hypothesis. We choose x and get I_3:

$$I_3 := \left\{ \begin{array}{ccc} (\{x \mapsto x_1, y \mapsto \mathtt{s}(y_0)\}, & \{\{x \mapsto x_1, y \mapsto y_0\}\}, & \emptyset), \\ (\{y \mapsto \mathtt{0}\}, & \emptyset, & \emptyset) \end{array} \right\}$$

Then, the new union is the following:

$$I_{1\cup 3} := \left\{ \begin{array}{ccc} (\{x \mapsto \mathtt{s}(x_0), y \mapsto \mathtt{0}\}, & \{\{x \mapsto x_0\}\}, & \emptyset), \\ (\{x \mapsto \mathtt{0}, y \mapsto \mathtt{s}(y_0)\}, & \{\{x \mapsto \mathtt{0}, y \mapsto y_0\}\}, & \emptyset), \\ (\{x \mapsto \mathtt{s}(x_0), y \mapsto \mathtt{s}(y_0)\}, & \{\{x \mapsto x_0\}, \{x \mapsto \mathtt{s}(x_0), y \mapsto y_0\}\}, & \emptyset), \\ (\{x \mapsto \mathtt{0}, y \mapsto \mathtt{0}\}, & \emptyset, & \emptyset) \end{array} \right\}$$

Now, we show well-foundedness of this by first removing x and the relations where its mapped values change from superterm to strict subterm. After removing duplicates, we have exactly I_2:

$$\left\{ \begin{array}{ccc} (\{y \mapsto \mathtt{0}\}, & \emptyset, & \emptyset), \\ (\{y \mapsto \mathtt{s}(y_0)\}, & \{\{y \mapsto y_0\}\}, & \emptyset), \end{array} \right\}$$

Even in cases where the induction terms of two induction schemes are identical, we may get a union that is not well-founded. As discussed earlier, having distinct induction terms can be a sign that the two induction schemes can be applied one after the other without "losing" anything. Also, the calculation of unions may consume a large amount of resources, not to mention the size of the induction schemes it gives when we have lots of schemes to begin with. Therefore, we only try to calculate the union of two induction schemes when their induction term sets have a non-empty intersection. We propose two heuristics for doing so. The first is a less aggressive heuristic:

Heuristic 3.8.1 (Union heuristic 1.) Given two induction schemes with induction term sets T_1 and T_2, we calculate their union if $T_1 \subseteq T_2$.

The second calculates the union if they have any intersection:

Heuristic 3.8.2 (Union heuristic 2.) Given two induction schemes with induction term sets T_1 and T_2, we calculate their union if $T_1 \cap T_2 \neq \emptyset$.

In general, as though multiple induction schemes with an intersecting set of induction terms may succeed separately, if we can merge them, the relation corresponding to the merged induction scheme subsumes all the original relations by construction, therefore we can always apply them and get at least as many matched hypothesis-conclusion pairs. Nonetheless, the question of whether merging schemes helps in terms of proof size or whether we can always obtain a well-founded union by weakening induction hypothesis as in the previous example lies outside of the scope of this thesis.

3.9 Constructor and destructor style induction

While using inductive types, one usually has to deal with *constructing* complex terms from simpler ones and *destructing* complex terms into simpler terms. Lots of recursive functions recurse into the arguments of certain inductive constructor terms such as `add` or `leq` for the type `nat`, `app` for `list`, while others may construct more complex terms and recurse on them. In particular, we are dealing with recursive function definitions where at least one argument is always changing to one of its subterms in any recursive call.

We can reference a complex term based on its subterms using constructor terms, e.g. the successor of x with $\mathtt{s}(x)$ or the binary tree containing two subtrees u and w and a value at the root v with $\mathtt{node}(u, v, w)$. Most of the recursive functions we have introduced so far contain constructor terms. However, we can also use destructor terms to reference the elements of a complex term. For instance, we can use equivalently instead of the pair x and $\mathtt{s}(x)$, the pair $\mathtt{p}(y)$ and y, where `p` is the predecessor function. Here we have an additional requirement that y cannot be `0` by mutual exclusion of term algebra constructors.

Converting terms from one style to another is always possible, from constructor to destructor terms we need to add conditions imposed by the mutual exclusiveness of term algebra constructors and from destructor to constructor terms we need to add new variables for not yet referenced arguments of the newly created constructor term.

Example 3.9.1 The following formula states that the first element of a non-empty list is in the list:

$$\forall x, y.\mathtt{in\_set}(x, \mathtt{cons}(x, y))$$

To convert this to destructor style, we replace $\mathtt{cons}(x, y)$ with a new variable z, all its subterms by appropriate destructor terms and add the side condition stating that z is not `nil`:

$$\forall z.z \neq \mathtt{nil} \to \mathtt{in\_set}(\mathtt{head}(z), z)$$

The next formula states that the left subtree of a binary tree flattened to a list (via the function `fltn`) is the prefix of the whole flattened tree:

$$\forall u.u \neq \mathtt{leaf} \to \mathtt{prefix}(\mathtt{fltn}(\mathtt{left}(u)), \mathtt{fltn}(u))$$

We can convert this to constructor style similarly by replacing every occurrence of u with a new term $\mathtt{node}(u_0, u_1, u_2)$ with fresh variables u_0, u_1 and u_2 and also, $\mathtt{left}(u)$ with u_0. The side condition becomes trivially true and can be removed:

$$\forall u_0, u_1, u_2.\mathtt{prefix}(\mathtt{fltn}(u_0), \mathtt{fltn}(\mathtt{node}(u_0, u_1, u_2)))$$

Both formats have their advantages and disadvantages: while the destructor style format contains less variables when we have constructors with greater arities and irrelevant

arguments, it creates bigger formulas due to the side conditions. We prefer to use constructor style definitions as they fit the superposition calculus better (although this has to be verified empirically) and we also find it easier to admit the well-foundedness criterion with constructors. For more details, see [KRV16].

3.10 Other techniques for combining induction schemes

We will now give a short survey of other techniques used in inductive theorem provers that combine induction schemes. One prominent example of a theorem prover using more advanced techniques for inductive reasoning is ACL2 [BM79, BM88]. Essentially, lots of the techniques used nowadays were implemented first in this theorem prover such as *recursion analysis* which is one of the main sources of motivation for our induction formula generation. We will now discuss in particular *subsumption* and *merging* of induction schemes as implemented in ACL2, moreover *flawed* and *unflawed* induction schemes.

There are some differences between the methods described earlier and the ones used by ACL2. One is that ACL2 uses only destructor style function definitions which prevents us from directly comparing our aforementioned techniques to the ones in ACL2. Another one is that all formulas in ACL2 are implicitly universally quantified which does not allow universal quantification inside the formulas, thus preventing strengthening of induction hypotheses.

Subsumption of induction schemes, in broad terms, checks whether every recursive case in the case distinction of an induction scheme can be injectively mapped to that of another scheme [MW13]. The mapping is done by comparing the side condition sets of two cases first and checking for subset containment, then checking whether the induction terms in the (destructor style) induction hypotheses can be injectively mapped by mapping one induction hypothesis to another if its a subterm of the other. E.g. consider these two recursive cases inducting on the variable x:

$$x \neq \mathtt{0} \rightarrow (P(\mathtt{p}(x)) \rightarrow P(x))$$

$$(x \neq \mathtt{0} \wedge \mathtt{p}(x) \neq \mathtt{0}) \rightarrow (P(\mathtt{p}(\mathtt{p}(x))) \rightarrow P(x))$$

Here, the first one is subsumed by the second as the condition set $\{x \neq \mathtt{0}\}$ is the subset of $\{x \neq \mathtt{0}, \mathtt{p}(x) \neq \mathtt{0}\}$ and $\mathtt{p}(x)$ is a subterm of $\mathtt{p}(\mathtt{p}(x))$. This means a case is subsumed by the other if the conditions of the latter imply that of the former and moreover, they "peel off" the same values from the inductive variables, the latter possibly more. Converting this idea to constructor style, we find that the cases correspond to these:

$$P(x) \rightarrow P(\mathtt{s}(x))$$

$$P(x) \rightarrow P(\mathtt{s}(\mathtt{s}(x)))$$

One "steps forward" one successor at a time, the other two. The reason this works well in ACL2 is probably partly due to the type system: with no arbitrary types and no

types with more than one non-base constructor, we can only recurse according to the only non-base constructor for any inductive type meaning any function that matches the innermost constructor will probably match the outer constructor terms as well. In the above example, the function that suggested to induct according to $\mathtt{s}(x)$ will also match $\mathtt{s}(\mathtt{s}(x))$ by applying the function twice. This subsumption gives the same results as simply checking containment in some cases but in others it also finds schemes subsumed by one another where containment in the sense of relations does not hold. As described in by Walther [Wal92], the side condition check has no theoretical basis as the implication is the other way around because $x \neq \mathtt{0} \wedge \mathtt{p}(x) \neq \mathtt{0}$ implies $x \neq \mathtt{0}$ and not vice versa. The subterm check in his analysis is simply checking whether one relation is in the transitive closure.

Induction scheme merging in ACL2 works by taking only the *intersection* of the induction schemes as described through an example of transitivity of `leq` in Section 3.8. As already mentioned, the intersection is contained by the composing schemes and therefore its use is theoretically not well-established. We once again refer to the analysis of Walther which describes the intersection as a necessary heuristic in ACL2 to compensate for the lack of universal quantification *inside* the induction formula. Otherwise, with strengthened induction hypotheses one can instantiate variables to the appropriate terms that are not present in the selected induction scheme.

Finally, we mention how induction schemes are categorized into flawed and unflawed ones by ACL2. A flawed induction scheme is not subsumed by others nor can be merged into others but has a non-empty intersection of induction terms with other schemes. Those schemes are disregarded and not used. This is partly done to solve the problems described above, thus disallowing any induction which would fail due to weak induction hypotheses. From the unflawed induction schemes, ACL2 selects the best according to some measurement and uses only that selected one. In saturation-based theorem proving, it makes no sense to only induct according to one scheme, instead of prioritizing them, we add all to the search space, as will be described in Chapter 4.

3.11 Selecting induction terms

Although each recursive function term in a formula suggests an induction scheme, in general not all such suggestions are good candidates for induction. For example, the term $\mathtt{add}(x, \mathtt{add}(y, z))$ contains two `add` function terms but only one suggests a good induction formula. This is because if we induct on say, y from the term $\mathtt{add}(y, z)$, the induction step will contain a term $\mathtt{add}(x, \mathtt{add}(\mathtt{s}(w), z))$, which can be simplified to $\mathtt{add}(x, \mathtt{s}(\mathtt{add}(w, z)))$ which does not match the term $\mathtt{add}(x, \mathtt{add}(w, z))$ in the induction hypothesis.

In order to use induction schemes only from "good" positions, we extend active argument positions used in function definitions to arbitrary complex terms and atomic formulas in the following way.

Definition 3.11.1 Given a term t in position p, we define its **active term positions** inductively based on its structure. If p is not an active term position, none of t's subterms are in active term positions. Otherwise, we distinguish these cases:

- if t is of the form $\mathtt{f}(s_1, ..., s_n)$ for some recursive function $\mathtt{f}$, pq for all $q \in I^a_{\mathtt{f}}$ are active term positions
- if t is of the form $\mathtt{c}(s_1, ..., s_m)$ for some constructor $\mathtt{c}$ of inductive type τ, all pr s.t. s_r is of type τ, for $1 \leq r \leq m$ are active term positions
- if t is of the form $g(s_1, ..., s_l)$ where g is some non-recursive (or uninterpreted) function term, all its argument positions $p1, ..., pl$ are active term positions.

Moreover, both top level positions (1 and 2) of an equality $l = r$ are active term positions and a literal $\mathtt{q}(t_1, ..., t_n)$ has active term positions $I^a_{\mathtt{q}}$ if $\mathtt{q}$ is a recursive predicate definition, otherwise all positions $1, ..., n$ are active term positions.

An uninterpreted or non-recursive function may have no case distinction at all, so it is not necessarily simplified. We therefore cannot hope for this term to disappear and since we potentially do not know anything about its structure, we end up better trying all its positions as active ones. These rules of course do not guarantee that a successful induction will be performed and there may be subterms which can be simplified to get more promising terms for induction. We leave the handling of simplification during induction to the prover implementation without further investigating this issue here. For more details, see Chapter 4.

Each of the function terms in active term positions suggests induction schemes if we can select one active subterm in all of their active argument positions (see Definition 3.4.3). For each active argument position, there may be multiple possibilities including non-variable terms. In principle, any such term can be selected for induction. It might be the case that such terms contain each other or they may be in distinct positions. The first case will be dealt with in the next section. For the second case, consider the following example:

Example 3.11.1 Given the function definition flattening a binary tree into a list:

$$
\begin{aligned}
&\textbf{fun } \mathtt{fltn}(x : \mathtt{btree}) \to \mathtt{list} := \\
&\quad \textbf{match } x \textbf{ with} \\
&\qquad (\mathtt{leaf} \to \mathtt{nil}) \\
&\qquad (\mathtt{node}(u, v, w) \to \mathtt{app}(\mathtt{fltn}(u), \mathtt{cons}(v, \mathtt{fltn}(w))))
\end{aligned}
$$

We would like to prove the conjecture:

$$\forall x, y, z, u, v.\mathtt{fltn}(\mathtt{node}(x, y, \mathtt{node}(z, u, v))) = \mathtt{fltn}(\mathtt{node}(\mathtt{node}(x, y, z), u, v))$$

This conjecture has multiple variables in active term positions, namely all occurrences of x, z and v and we may induct on any subset of these. Here, we do not know whether any of these combinations will succeed without looking into the function definition or actually simplifying the conjecture. After simplification, the conjecture becomes:

$$\forall x, y, z, u, v.\mathtt{app}(\mathtt{fltn}(x), \mathtt{cons}(y, \mathtt{app}(\mathtt{fltn}(z), \mathtt{cons}(u, \mathtt{fltn}(v))))) = \mathtt{app}(\mathtt{app}(\mathtt{fltn}(x), \mathtt{cons}(y, \mathtt{fltn}(z))), \mathtt{cons}(u, \mathtt{fltn}(v)))$$

At this point, the set of variables in active term positions reduces to only x.

For cases like these, where simplification is possible or there is more than one term that we can select for an active argument position of a function, we are tempted not to create any induction schemes and leave the simplification to the prover. In most cases, this helps as we reduce the number of inductions while still keeping the necessary ones, however, for some exampes it is still beneficial to try to induct on non-simplified formulas (see Chapter 6)

Another issue arises when we have multiple occurrences of a term but not all of them are in active term positions. Consider the conjecture:

$$\forall x.\mathtt{add}(x, \mathtt{add}(x, x)) = \mathtt{add}(\mathtt{add}(x, x), x)$$

Inducting on all occurrences of x will result in a step case like:

$$\forall z.\mathtt{add}(z, \mathtt{add}(z, z)) = \mathtt{add}(\mathtt{add}(z, z), z) \rightarrow \mathtt{add}(\mathtt{s}(z), \mathtt{add}(\mathtt{s}(z), \mathtt{s}(z))) = \mathtt{add}(\mathtt{add}(\mathtt{s}(z), \mathtt{s}(z)), \mathtt{s}(z))$$

Simplification results in the formula:

$$\forall z.\mathtt{add}(z, \mathtt{add}(z, z)) = \mathtt{add}(\mathtt{add}(z, z), z) \rightarrow \mathtt{s}(\mathtt{add}(z, \mathtt{s}(\mathtt{add}(z, \mathtt{s}(z))))) = \mathtt{s}(\mathtt{add}(\mathtt{add}(z, \mathtt{s}(z)), \mathtt{s}(z)))$$

At this point we are obviously stuck as the induction hypothesis does not match the conclusion.

This issue can be solved by *generalizing over the occurrences* of x. Generalization means that we create a more general conjecture instead of the original and try to prove that. We can replace any number of occurrences of the original variables with fresh variables as long as we do not use the same fresh variables to replace different original variables. In our example, instead of selecting all occurrences of x, we select only the ones in active term positions and generalize over those with fresh variables, which results in the

$$\begin{array}{l}\textbf{fun } \mathtt{dup}(x : \mathtt{nat}) \to \mathtt{nat} := \\ \quad \textbf{match } x \textbf{ with} \\ \quad\quad (\mathtt{0} \to \mathtt{0}) \\ \quad\quad (\mathtt{s}(z) \to \mathtt{s}(\mathtt{s}(\mathtt{dup}(z))))\end{array}$$

Figure 3.4: The function `dup`

following induction formula:

$$\forall x. \left(\left(\forall z. \begin{pmatrix} \mathtt{add}(\mathtt{0}, \mathtt{add}(x,x)) = \mathtt{add}(\mathtt{add}(\mathtt{0},x),x) \wedge \\ \mathtt{add}(z, \mathtt{add}(x,x)) = \mathtt{add}(\mathtt{add}(z,x),x) \to \\ \mathtt{add}(\mathtt{s}(z), \mathtt{add}(x,x)) = \mathtt{add}(\mathtt{add}(\mathtt{s}(z),x),x) \end{pmatrix} \right) \right) \\ \to \forall w. \mathtt{add}(w, \mathtt{add}(x,x)) = \mathtt{add}(\mathtt{add}(w,x),x)$$

If the generalized conjecture can be proven, the original is also proven since it is a special case of the generalized one. If we cannot prove the generalized, it might be that we *overgeneralized* our conjecture, rendering it a non-theorem.

Other cases, like the commutativity of `add` suggests selecting only the active positions is not always a good idea, since it generalizes to a non-theorem e.g. by replacing only the active occurrences of x with a new variable z:

$$\forall x, y, z. \mathtt{add}(z, y) = \mathtt{add}(y, x)$$

This formula is obviously false so we cannot hope to prove it. The heuristic described in [Cru17] selects all occurrences of a term if only one active or one non-active occurrence is present. This solves also problems like the following with the definition for `dup` shown in Figure 3.4:

$$\forall x. \mathtt{dup}(x) = \mathtt{add}(x, x)$$

Selecting only the active occurrences – the first and second – results in a non-theorem so it is necessary to select the third one too. However, this heuristic does not work with theorems like this:

$$\forall x. \mathtt{leq}(x, \mathtt{add}(x, x))$$

In the end, we find that a relatively simple heuristic comes at the price of working on a certain type of problems but not on others. We propose to use two heuristics. The first heuristic is the following:

Heuristic 3.11.1 (Generalized induction term occurrence selection 1.) From each literal we select only the active occurrences of an induction term for generalization if the number of active occurrences is at least two. Otherwise we select all occurrences.

And the second:

Heuristic 3.11.2 (Generalized induction term occurrence selection 2.) From each literal we select only the active occurrences of an induction term for generalization if the number of active occurrences is at least two and the number of non-active occurrences is at least two. Otherwise we select all occurrences.

We will later look at how to use both heuristics incorporated into different strategies to be able to solve more problems.

3.12 Complex terms as induction terms

As described in Section 3.11, not only variables can be selected as induction terms while creating induction schemes. In fact, we can induct on any term and get a valid induction formula. This is also a kind of generalization – a very powerful one –, where we get rid of some specific subterm structure of a conjecture which we suspect is irrelevant to our proof, thus getting a more general conjecture by replacing a number of occurrences of this subterm with variables in the induction formula.

For instance, consider the following conjecture:

$$\forall x, y, z, u, v, w.\mathtt{add}(\mathtt{add}(x, \mathtt{add}(y, \mathtt{add}(z, u))), \mathtt{add}(v, w)) = \\ \mathtt{add}(\mathtt{add}(\mathtt{add}(x, \mathtt{add}(y, \mathtt{add}(z, u))), v), w)$$

This can be proven by first inducting on x, then after simplifying the subformulas, inducting on y, etc. Otherwise we can notice that both sides contain the subterm $\mathtt{add}(x, \mathtt{add}(y, \mathtt{add}(z, u)))$ which we can throw away and replace with a fresh variable x_0 to get the more general and still provable conjecture:

$$\forall x_0, v, w.\mathtt{add}(x_0, \mathtt{add}(v, w)) = \mathtt{add}(\mathtt{add}(x_0, v), w)$$

We note here that the only subterm of the generalized subterm that can be inducted on (at least by our methods described so far) is x since none of the other subterms are in active term positions. So to avoid overgeneralizing a conjecture, we must utilize all possible induction terms and try all corresponding induction formulas simultaneously.

This can significantly increase the size of the search space. Therefore, we have to be careful how we select the complex terms for induction. First of all, only complex terms in active term positions can be selected. This helps in some cases, while in others it still means all terms are selected. One idea is to induct on a complex term if it has more than one occurrence. Another one is to only induct on a complex term t if none of its subterms show up outside of the occurrences of t.

Example 3.12.1 Consider the following conjecture:

$$\forall x, y.\mathtt{app}(\mathtt{fltn}(x), y) = \mathtt{fltn}_2(x, y)$$

$$\begin{aligned}&\textbf{fun } \texttt{fltn}_2(x : \texttt{btree}, y : \texttt{list}) \to \texttt{list} := \\ &\quad \textbf{match } x \textbf{ with} \\ &\qquad (\texttt{leaf} \to y) \\ &\qquad (\texttt{node}(x_1, x_2, x_3) \to (\texttt{fltn}_2(x_1, \texttt{cons}(x_2, \texttt{fltn}_2(x_3, y)))))\end{aligned}$$

Figure 3.5: The function $\texttt{fltn}_2$

$\texttt{fltn}_2$ is the *fold* alternative definition for $\texttt{fltn}$ shown in Figure 3.5. First, we induct on x with the following induction formula with strengthened hypotheses:

$$\forall y. \left(\left(\forall x_0, x_1, x_2. \left(\begin{pmatrix} \texttt{app}(\texttt{fltn}(\texttt{leaf}), y) = \texttt{fltn}_2(\texttt{leaf}, y) \wedge \\ \begin{pmatrix} \forall y_0.\texttt{app}(\texttt{fltn}(x_0), y_0) = \texttt{fltn}_2(x_0, y_0) \wedge \\ \forall y_1.\texttt{app}(\texttt{fltn}(x_2), y_1) = \texttt{fltn}_2(x_2, y_1) \end{pmatrix} \to \\ \texttt{app}(\texttt{fltn}(\texttt{node}(x_0, x_1, x_2)), y) = \texttt{fltn}_2(\texttt{node}(x_0, x_1, x_2), y) \end{pmatrix} \right) \right) \right)$$
$$\to \forall w.\texttt{app}(\texttt{fltn}(w), y) = \texttt{fltn}_2(w, y)$$

The base case can be easily solved by just applying the definitions of the three functions. The recursive case consequent can be simplified to get:

$$\texttt{fltn}_2(x_0, \texttt{cons}(x_1, \texttt{fltn}_2(x_2, y))) = \texttt{app}(\texttt{app}(\texttt{fltn}(x_0), \texttt{cons}(x_1, \texttt{fltn}(x_2))), y)$$

Now we can apply the strengthened induction hypotheses twice to get rid of $\texttt{fltn}_2$ completely:

$$\texttt{app}(\texttt{fltn}(x_0), \texttt{cons}(x_1, \texttt{app}(\texttt{fltn}(x_2), y))) = \texttt{app}(\texttt{app}(\texttt{fltn}(x_0), \texttt{cons}(x_1, \texttt{fltn}(x_2))), y)$$

We get an equality with noticeably similar structure on both sides. However, we get stuck here as no more simplification can be done and if we induct once again, now on x_0, we get an even greater equality with the same problems. To resolve this issue, we need to induct on $\texttt{fltn}(x_0)$. Notice that this term has only two occurrences – both in active term positions – and its only proper subterm x_0 has no occurrence outside this term. These conditions make $\texttt{fltn}(x_0)$ a very good candidate for induction. We omit the whole induction formula for readability. The base case can be proven by applying simplifications:

$$\texttt{app}(\texttt{nil}, \texttt{cons}(x_1, \texttt{app}(\texttt{fltn}(x_2), y))) = \texttt{app}(\texttt{app}(\texttt{nil}, \texttt{cons}(x_1, \texttt{fltn}(x_2))), y)$$

Next is the recursive case consequent:

$$\begin{aligned}&\texttt{app}(\texttt{cons}(u_0, u_1), \texttt{cons}(x_1, \texttt{app}(\texttt{fltn}(x_2), y))) = \\ &\quad \texttt{app}(\texttt{app}(\texttt{cons}(u_0, u_1), \texttt{cons}(x_1, \texttt{fltn}(x_2))), y)\end{aligned}$$

$$\begin{aligned}&\textbf{fun } \mathtt{mul}(x : \mathtt{nat}, y : \mathtt{nat}) \to \mathtt{nat} := \\ &\quad \textbf{match } x \textbf{ with} \\ &\quad\quad (\mathtt{0} \to \mathtt{0}) \\ &\quad\quad (\mathtt{s}(x_0) \to \mathtt{add}(\mathtt{mul}(x_0, y), y))\end{aligned}$$

Figure 3.6: The function `mul`

Simplification applied on this results in the equality:

$$\begin{aligned}&\mathtt{cons}(u_0, \mathtt{app}(u_1, \mathtt{cons}(x_1, \mathtt{app}(\mathtt{fltn}(x_2), y)))) = \\ &\quad \mathtt{cons}(u_0, \mathtt{app}(\mathtt{app}(u_1, \mathtt{cons}(x_1, \mathtt{fltn}(x_2))), y))\end{aligned}$$

This reduces to just the second arguments of the `cons` terms by term algebra injectivity:

$$\mathtt{app}(u_1, \mathtt{cons}(x_1, \mathtt{app}(\mathtt{fltn}(x_2), y))) = \mathtt{app}(\mathtt{app}(u_1, \mathtt{cons}(x_1, \mathtt{fltn}(x_2))), y)$$

Finally, as this equality is exactly the induction hypothesis, we are done. Notice that in the second induction on $\mathtt{fltn}(x_0)$, we effectively got rid of all active occurrences of the `fltn` symbol and we solved a special case of `app` associativity where the rest of `fltn` symbols had no active role.

Only selecting complex terms when none of its variables can be found outside this particular term is however, in some cases too restrictive. Consider the following example:

Example 3.12.2 Given the function definition for multiplication in Figure 3.6, prove its associativity:

$$\forall x, y, z.\mathtt{mul}(\mathtt{mul}(x, y), z) = \mathtt{mul}(x, \mathtt{mul}(y, z))$$

First, we induct on x with the following induction formula:

$$\forall y, z. \left(\left(\begin{array}{c} \mathtt{mul}(\mathtt{mul}(\mathtt{0}, y), z) = \mathtt{mul}(\mathtt{0}, \mathtt{mul}(y, z)) \wedge \\ \forall x_0. \left(\begin{array}{c} \mathtt{mul}(\mathtt{mul}(x_0, y), z) = \mathtt{mul}(x_0, \mathtt{mul}(y, z)) \to \\ \mathtt{mul}(\mathtt{mul}(\mathtt{s}(x_0), y), z) = \mathtt{mul}(\mathtt{s}(x_0), \mathtt{mul}(y, z)) \end{array} \right) \\ \to \forall x.\mathtt{mul}(\mathtt{mul}(x, y), z) = \mathtt{mul}(x, \mathtt{mul}(y, z)) \end{array} \right) \right)$$

The base case is trivial. Simplifying the step case consequent, we get:

$$\mathtt{mul}(\mathtt{add}(\mathtt{mul}(x_0, y), y), z) = \mathtt{add}(\mathtt{mul}(x_0, \mathtt{mul}(y, z)), \mathtt{mul}(y, z))$$

We cannot simplify it further and there are no complex terms shared by the two sides to induct on but we can rewrite the first argument of the `add` term on the right-hand

side with the induction hypothesis:

$$\texttt{mul}(\texttt{add}(\texttt{mul}(x_0, y), y), z) = \texttt{add}(\texttt{mul}(\texttt{mul}(x_0, y), z), \texttt{mul}(y, z))$$

Here, inducting on simply x_0 leads to a dead-end where we get an even more complex structure. Instead, we notice that $\texttt{mul}(x_0, y)$ is present on both sides. This is promising, as inducting on this will result in a simpler generalized subgoal but y also appears outside of $\texttt{mul}(x_0, y)$. This is no problem as the two remaining ys can be paired up on the two sides of the equality and are not affected by the generalization.

We omit the full proof as it contains one more induction and solving a commutativity subgoal for `add`.

In summary, the complex term induction heuristic we use is the following:

Heuristic 3.12.1 (Generalized complex induction term occurrence selection) We induct on complex terms in active term positions that have at least one other occurrence in the same inductive goal. We select occurrences based on the term occurrence selection heuristic used for variables.

3.13 The induction heuristic

We conclude this chapter by presenting a general algorithmic approach – a heuristic – to tackle inductive problems. Given a formula F and a set of induction templates, the steps are the following:

1. Find all atomic subformulas (literals) in F.
2. Look for function terms in active term positions and non-equality predicate terms in these literals that have an induction template based on Section 3.11.
3. For each of these terms with function `f`, collect all possible selected term tuples in positions $I^a_{\texttt{f}}$ in active subterm positions (Section 3.11).
4. Instantiate induction schemes from the selected term tuples with their corresponding induction template and function/predicate terms (Definition 3.4.2).
5. Discard induction schemes with complex terms based on the complex term heuristic 3.12.1.
6. Check containment of induction schemes based on Section 3.6. If an induction scheme is contained by another, discard it.
7. Calculate the union of the rest of the schemes if suggested by the current union heuristic based on Section 3.8. If the union is well-founded, add it to the set of induction schemes and discard the original ones. Do this until nothing changes.

8. Instantiate induction formulas from F based on the set of induction schemes (Section 3.4.1). While doing so, replace induction term occurrences in the instantiated formulas according to the current induction term occurrence heuristic, see Section 3.11. Strengthen induction hypotheses with fresh variables if needed (Section 3.7).

9. Simplify the new formulas and repeat on each from step 1.

CHAPTER 4

Induction in proof search

So far we described induction in a general setting. In this chapter, first we show how induction is applied in a saturation-based theorem prover, then using our results from the previous chapter, we describe additional methods and address some of the difficulties in this new setting.

4.1 Induction inference

In saturation-based proof search, a universally quantified inductive goal $\forall x.L[x]$ with one variable x and one literal L is negated and Skolemized resulting in the ground formula $\neg L[\sigma]$. To prove this goal inductively, an induction formula is generated for this goal – which is just any valid formula of the form:

$$formula \rightarrow \forall x.L[x]$$

This is then clausified resulting in clauses that contain the consequent $\forall x.L[x]$ of the induction formula positively. Each such clause can be directly binary resolved with the negated goal. This idea is captured in the following inference rule which is a more general form using any clause $L[\sigma_1]...[\sigma_n] \vee C$ as the main premise [RV19]:

$$\frac{formula \rightarrow \forall x_1, ..., x_n.\overline{L}[x_1]...[x_n] \qquad L[\sigma_1]...[\sigma_n] \vee C}{\texttt{cnf}(\neg formula \vee C)}\ (\texttt{Ind})$$

where $L[\sigma_1]...[\sigma_n]$ is a ground literal and $formula \rightarrow \forall x_1, ..., x_n.\overline{L}[x_1]...[x_n]$ is a valid induction formula.

In saturation-based proof search it would also make sense to just add the clausified induction formula to the search space and let the prover resolve it against the goals it applies to but then it would take much more time or perhaps it would not even happen

before resource exhaustion. Therefore, we choose this more goal-oriented approach and force binary resolution during the application of the inference.

Theorem 4.1.1 The inference rule `Ind` is sound.

Proof. Since the left premise of the rule is a valid induction formula and since binary resolution is sound, we just need to check whether the conditions apply to be able to resolve the two premises. By construction, the main consequent of the induction formula $\forall x_1, ..., x_n.\overline{L}[x_1]...[x_n]$ only contains the fresh variables x_i where there is a Skolem constant σ_i in the right premise and as the right premise is ground, we can apply binary resolution with the substitution $\theta = \{x_i \mapsto \sigma_i \mid 1 \leq i \leq n\}$, so the consequence of the rule follows from the premises. □

The inference rule `Ind` is sound as long as the generated induction formula is valid, meaning the order defined by the relations between the induction hypotheses and consequents are well-founded and the remaining base cases complete the case distinction for the induction terms, as described in the previous chapter.

The induction heuristic and the techniques we presented in Chapter 3 differ in a few points. One difference is that we do not have any variables in the subgoal we are considering, since it is ground – we have to select Skolem constants or other (possibly complex) ground terms as induction terms. Another difference is that we have to generate the induction formula based on the *opposite sign literal* to get the right resolving before achieving the result. Hence, we also have to consider disequalities the same way when looking for active term positions. Let us look at an example:

Example 4.1.1 Consider the append and prefix functions for type `list` described by the following function definitions:

$$\begin{array}{l}\textbf{fun}\ \texttt{app}(x : \texttt{list}, y : \texttt{list}) \to \texttt{list} := \\ \quad \textbf{match}\ x\ \textbf{with} \\ \quad\quad (\texttt{nil} \to y) \\ \quad\quad (\texttt{cons}(x_0, z) \to \texttt{cons}(x_0, \texttt{app}(z, y)))\end{array}$$

$$\begin{array}{l}\textbf{fun}\ \texttt{pref}(x : \texttt{list}, y : \texttt{list}) \to \texttt{bool} := \\ \quad \textbf{match}\ x\ \textbf{with} \\ \quad\quad (\texttt{nil} \to \texttt{true}) \\ \quad\quad (\texttt{cons}(x_0, z) \to\ \textbf{match}\ x\ \textbf{with} \\ \quad\quad\quad (\texttt{nil} \to \texttt{false}) \\ \quad\quad\quad (\texttt{cons}(y_0, u) \to x_0 = y_0 \wedge \texttt{pref}(z, u)))\end{array}$$

and the following conjecture:

$$\forall x, y.\mathtt{pref}(x, \mathtt{app}(x, y))$$

This conjecture is refuted by first negating and Skolemizing it to get:

$$\neg\mathtt{pref}(\sigma_0, \mathtt{app}(\sigma_0, \sigma_1))$$

We use the techniques from Chapter 3, specifically, apply the induction heuristic step by step presented in Section 3.13. Using the induction templates of `pref` and `app`, we create induction schemes based on the function terms in active positions. In particular, the induction term σ_0 in active term positions (i.e. both positions) give two identical schemes. After discarding one of them, we can generate an induction formula from the one scheme and get the following:

$$\left(\begin{array}{c}\mathtt{pref}(\mathtt{nil}, \mathtt{app}(\mathtt{nil}, \sigma_1)) \quad \wedge \\ \forall x_0, z.\left(\begin{array}{c}\mathtt{pref}(z, \mathtt{app}(z, \sigma_1)) \quad \rightarrow \\ \mathtt{pref}(\mathtt{cons}(x_0, z), \mathtt{app}(\mathtt{cons}(x_0, z), \sigma_1))\end{array}\right)\end{array}\right) \rightarrow \forall x.\mathtt{pref}(x, \mathtt{app}(x, \sigma_1))$$

As we can see, the induction formula contains the original goal, i.e. the negation of the literal we selected. The result is then clausified with all clauses resolved with the negated goal if possible:

$$\left\{\begin{array}{ll}\{\neg\mathtt{pref}(\mathtt{nil}, \mathtt{app}(\mathtt{nil}, \sigma_1)), & \mathtt{pref}(x, \mathtt{app}(x, \sigma_1))\}, \\ \{\neg\mathtt{pref}(\mathtt{nil}, \mathtt{app}(\mathtt{nil}, \sigma_1)), & \neg\mathtt{pref}(\mathtt{cons}(x_0, x), \mathtt{app}(\mathtt{cons}(x_0, x), \sigma_1))\}\end{array}\right\}$$

Note that we use this rule as the basis for our inferences as it fits our purposes best, a discussion on this and other induction inferences rules can be found in [RV19, HHK+20].

4.2 Rewriting with function definitions

When an input function definition of the form **fun** $\mathtt{f}(x_1, ..., x_n) := \mathit{body}$ is given to a saturation-based theorem prover, besides using the function definition for generating induction formulae, it needs to be used also by the superposition calculus inference rules. This is usually done by folding out the definition in a preprocessing step to m branches of the function given by `match` and `ite` blocks to get formulas of the form

$$F_1 \rightarrow \mathtt{f}(x_1, ..., x_n) = t_1$$

$$...$$

$$F_m \rightarrow \mathtt{f}(x_1, ..., x_n) = t_m$$

where $F_1, ..., F_m$ are *side conditions* corresponding to the branches with body terms $t_1, ..., t_m$. Then, each formula is converted to clausal form. A special case comes with

functions of type `bool` as these are converted to equivalences:

$$\mathtt{f}(x_1, ..., x_n) \leftrightarrow (F_1 \rightarrow G_1)$$

...

$$\mathtt{f}(x_1, ..., x_n) \leftrightarrow (F_m \rightarrow G_m)$$

with $G_1, ..., G_m$ corresponding to the `bool` terms each of these branches contain.

While one essential ingredient during induction is being able to identify the active term positions of a function where we can place constructor terms that match one of the branches of that function, in order to use any of the branches for rewriting, we have to expand the *function headers into their definitions*. While this seems like given in a manual proof, in a superposition theorem prover one can only apply any inferences if the side conditions for the accompanying simplification ordering are satisfied.

In particular, a clause defining a single branch for a function `f` containing an equality

$$\mathtt{f}(x_1, ..., x_n) = t \vee C$$

cannot be used to rewrite any occurrence of `f` if the term t is heavier in the ordering than $\mathtt{f}(x_1, ..., x_n)$ or the literal $\mathtt{f}(x_1, ..., x_n) = t$ is not selected in the clause. The choices we have here is either not being able to simplify induction formulas or losing completeness – this latter can cause e.g. looping.

One solution is to choose an ordering that orients all function definitions according to their intended usage. Although this task is in general undecidable [H+78], such an ordering can be found e.g. for KBO in polynomial time if it exists [DKM05, KV03]. However, even if we can find one before starting the proof search, there may be newly inferred clauses during the proof based on the initial set of function definitions that may be also used as function definitions. For example, using the recursive axiom for `add` and the term algebra exhaustiveness for `nat`, we can infer the following:

$$\frac{\forall x.x = \mathtt{0} \vee x = \mathtt{s}(\mathtt{p}(x)) \qquad \forall x, y.\mathtt{add}(\mathtt{s}(x), y) = \mathtt{s}(\mathtt{add}(x, y))}{\forall x, y.x = \mathtt{0} \vee \mathtt{add}(x, y) = \mathtt{s}(\mathtt{add}(\mathtt{p}(x), y))}\ (\mathtt{Sup})$$

The conclusion is actually the destructor-style definition of the recursive case of `add`, so intuitively one would use this also as a function definition. The orientation of any such inferred equality however might be different from the intended one in the current term ordering. This could be mitigated by adjusting the ordering during proof search which is not practically achievable. Otherwise, conflicting orientations of two variants for the same function definition can cause an infinite chain of rewriting.

Another solution is to allow rewriting according to the original function definition orientation (which might be different from the one given by the ordering) and at the same time disallow any inferences that would rewrite the function definitions themselves, creating new variants. Despite losing completeness in this case, depending on factors like

whether we use constructor- or destructor-style function definitions, it can actually help discard inferences which we would not be used anyways, e.g. any instantiated variant for a function definition just creates redundancy in the search space.

Our proposed solution is the following: we mark equalities stemming from function definitions. When we encounter a clause containing such a marked equality during saturation, we disallow any inferences with it except for rewriting with two special inference rules: one is a modified *paramodulation*, the other is a special case of demodulation. Paramodulation is a special case of superposition which only rewrites instances of the rewriting equality left-hand side:

$$\frac{l = r \vee C \qquad L[l\theta] \vee D}{L[r\theta] \vee C\theta \vee D}\ (\texttt{Par})$$

where θ is the mgu of l and r, moreover $l\theta \succ r\theta$, $l\theta = r\theta \succ C$ and $L[l\theta] \succ D$. Paramodulation is mostly omitted from inference systems as it is entirely subsumed by superposition. We modify these two rules as follows: first, *we let demodulation happen only if the function definition is a unit clause and it has an orientation which is simplifying w.r.t. the ordering.*

$$\frac{\mathtt{f}(\overline{x}) = t \qquad \cancel{C[\mathtt{f}(\overline{x})\theta] \vee D}}{C[t\theta] \vee D}\ (\texttt{DemF})$$

where $\mathtt{f}(\overline{x}) = t$ is a function definition, moreover $\mathtt{f}(\overline{x})\theta \succ t\theta$ and $C[\mathtt{f}(\overline{x})\theta] \vee D \succ \mathtt{f}(\overline{x})\theta = t\theta$.

Second, *all other non-simplifying rewriting with function definitions is done with the modified paramodulation rule*:

$$\frac{\mathtt{f}(\overline{x}) = t \vee C \qquad L[\mathtt{f}(\overline{x})\theta] \vee D}{L[t\theta] \vee C\theta \vee D}\ (\texttt{ParF})$$

where $\mathtt{f}(\overline{x}) = t$ is a function definition. Therefore we allow rewriting with function definitions that disregard the ordering or the literal selection so we can rewrite any part of a clause. This will help expand function headers when needed but at the same time we lose properties such as fairness or completeness.

Note that we only use clauses for such rewriting that contain exactly one equality literal marked as a function definition. One might define e.g. non-determinism with clauses containing more than one equality:

$$\mathtt{f}(\overline{x}) = t_1 \vee \mathtt{f}(\overline{x}) = t_2$$

We avoid handling such clauses in this manner as it is unclear how the other literal should be used in the resulting clause when we paramodulate with one of them.

Example 4.2.1 In Example 3.12.1, the two functions $\mathtt{fltn}$ and $\mathtt{fltn}_2$ give the four definitional axioms:

$$\mathtt{fltn}(\mathtt{leaf}) = \mathtt{nil}$$

$$\forall x, y, z.\mathtt{fltn}(\mathtt{node}(x, y, z)) = \mathtt{app}(\mathtt{fltn}(x), \mathtt{cons}(y, \mathtt{fltn}(z)))$$

$$\forall y.\mathtt{fltn}_2(\mathtt{leaf}, y) = y$$

$$\forall x, y, z, u.\mathtt{fltn}_2(\mathtt{node}(x, y, z), u) = \mathtt{fltn}_2(x, \mathtt{cons}(y, \mathtt{fltn}_2(z, u)))$$

The base cases for the two functions can be easily oriented the "right way", i.e. to rewrite any terms such as $\mathtt{fltn}(\mathtt{leaf})$ into $\mathtt{nil}$ or $\mathtt{fltn}_2(\mathtt{leaf}, y)$ into y, the first by making $\mathtt{fltn}$ or $\mathtt{leaf}$ greater than $\mathtt{nil}$ in the precedence accompanying the ordering, the second by applying the subterm property of a simplification ordering. The recursive cases, however, cannot be oriented in such a way easily, so e.g. the refutational variant of the induction step consequent from Example 3.12.1

$$\mathtt{app}(\mathtt{fltn}(\mathtt{node}(\sigma_0, \sigma_1, \sigma_2)), \sigma_3) \neq \mathtt{fltn}_2(\mathtt{node}(\sigma_0, \sigma_1, \sigma_2), \sigma_3)$$

would not be rewritten normally and the induction gets stuck. Instead, applying the rule (ParF) with the recursive axioms, we match the terms $\mathtt{fltn}(\mathtt{node}(\sigma_0, \sigma_1, \sigma_2)$ with $\mathtt{fltn}(\mathtt{node}(x, y, z))$ and $\mathtt{fltn}_2(\mathtt{node}(\sigma_0, \sigma_1, \sigma_2), \sigma_3)$ with $\mathtt{fltn}_2(\mathtt{node}(x, y, z), u)$ with substitution $\{x \mapsto \sigma_0, y \mapsto \sigma_1, z \mapsto \sigma_2, u \mapsto \sigma_3\}$.

4.3 Simplification and induction

Most saturation algorithms first apply simplification or even deletion rules to any clause to reduce the search space as much as possible, then if there is no more simplification to be done, start using generating inference rules to explore other parts of the search space. In this context, induction is a rule which generates a lot of clauses, therefore its application is usually at the very end of the saturation loop body.

This means that any clause that cannot be simplified but can be used in e.g. superposition with some function definition will be inducted on multiple times, once with the original clause and later with the generated new clauses. This only makes sense if induction and rewriting are not confluent, e.g. the order of their applications matter and result in different clause sets. On the other hand, it mostly just introduces redundancy in the set of clauses. Therefore, in one variant of our induction rule, we add a check not to induct on any literal that can be simplified or rewritten by some function definition. This is only preferred if we use the function definition rewriting rules from the previous section, otherwise we may not rewrite anything in the literal because the ordering prohibits it.

4.4 Multi-clause induction

The inference rule `Ind` works well for atomic goals, like Example 4.1.1. By generating an induction formula and resolving it with the goal, we essentially expand the goal into a term algebra case distinction based on some of its terms. This, however, does not work on more complex problems. Consider the following conjecture:

$$\forall x, y.(\texttt{even}(x) \wedge \texttt{even}(y)) \rightarrow \texttt{even}(\texttt{add}(x, y))$$

Now, negating this conjecture results in three clauses:

$$\{\{\texttt{even}(\sigma_0)\}, \{\texttt{even}(\sigma_1)\}, \{\neg\texttt{even}(\texttt{add}(\sigma_0, \sigma_1))\}\}$$

We can induct on e.g. the consequent of the original implication, using the induction scheme suggested by `even`:

$$\begin{pmatrix} \texttt{even}(\texttt{add}(0, \sigma_1)) \wedge \texttt{even}(\texttt{add}(\texttt{s}(0), \sigma_1)) \wedge \\ \forall x.\texttt{even}(\texttt{add}(x, \sigma_1)) \rightarrow \texttt{even}(\texttt{add}(\texttt{s}(\texttt{s}(x)), \sigma_1)) \end{pmatrix} \rightarrow \forall z.\texttt{even}(\texttt{add}(z, \sigma_1))$$

After using this with the inference `Ind`, we get the following:

$$\texttt{cnf}\left(\neg\left(\begin{array}{c} \texttt{even}(\texttt{add}(0, \sigma_1)) \wedge \texttt{even}(\texttt{add}(\texttt{s}(0), \sigma_1)) \wedge \\ \forall x.\left(\texttt{even}(\texttt{add}(x, \sigma_1)) \rightarrow \texttt{even}(\texttt{add}(\texttt{s}(\texttt{s}(x)), \sigma_1))\right) \end{array}\right)\right)$$

The clausification results in the following two clauses:

$$\left\{\begin{array}{lll} \{\neg\texttt{even}(\texttt{add}(0, \sigma_1)), & \neg\texttt{even}(\texttt{add}(\texttt{s}(0), \sigma_1)), & \texttt{even}(\texttt{add}(\sigma_2, \sigma_1))\}, \\ \{\neg\texttt{even}(\texttt{add}(0, \sigma_1)), & \neg\texttt{even}(\texttt{add}(\texttt{s}(0), \sigma_1)), & \neg\texttt{even}(\texttt{add}(\texttt{s}(\texttt{s}(\sigma_2)), \sigma_1))\} \end{array}\right\}$$

We use the principle of most saturation algorithms to *simplify clauses eagerly, generate new clauses lazily*, so we first simplify the clauses as much as we can using the definitions of `even` and `add`:

$$\left\{\begin{array}{lll} \{\neg\texttt{even}(\sigma_1), & \neg\texttt{even}(\texttt{s}(\sigma_1)), & \texttt{even}(\texttt{add}(\sigma_2, \sigma_1))\}, \\ \{\neg\texttt{even}(\sigma_1), & \neg\texttt{even}(\texttt{s}(\sigma_1)), & \neg\texttt{even}(\texttt{add}(\sigma_2, \sigma_1))\} \end{array}\right\}$$

Now, after all simplifications has been performed on this set of clauses, we can move on and generate new clauses. First, we can binary resolve both clauses with $\{\texttt{even}(\sigma_1)\}$:

$$\left\{\begin{array}{ll} \{\neg\texttt{even}(\texttt{s}(\sigma_1)), & \texttt{even}(\texttt{add}(\sigma_2, \sigma_1))\}, \\ \{\neg\texttt{even}(\texttt{s}(\sigma_1)), & \neg\texttt{even}(\texttt{add}(\sigma_2, \sigma_1))\} \end{array}\right\}$$

Then we can resolve the two clauses, matching the remains of the inductive step consequent with the induction hypothesis, resulting in this single clause:

$$\{\neg\texttt{even}(\texttt{s}(\sigma_1)), \neg\texttt{even}(\texttt{s}(\sigma_1))\}$$

After *duplicate literal removal*, we are stuck since we cannot get rid of this remaining literal:

$$\{\neg \texttt{even}(\texttt{s}(\sigma_1))\}$$

To understand what happened, let us look at the original conjecture as if we were performing the induction on it manually. To be more specific, in the case which we could not remove in the end we substituted $\texttt{s}(\texttt{0})$ for σ_0, which translates to substituting the same for x in the original conjecture:

$$\forall y.(\texttt{even}(\texttt{s}(\texttt{0})) \wedge \texttt{even}(y)) \rightarrow \texttt{even}(\texttt{add}(\texttt{s}(\texttt{0}), y))$$

This case can be proven very easily thanks to the false antecedent, since $\texttt{even}(\texttt{s}(\texttt{0}))$ is false. But the antecedents were not present in our induction, at least not in this form. Notice also how the translation of the induction cases to a clausal form replace the old Skolem constant σ_0 with a new one σ_2 in the induction step case, making it impossible to match any of the old clauses containing σ_0 with the new ones.

So the problem here is that although we need to make a case distinction on $\texttt{even}(\sigma_0)$ as well since it also contains the induction term σ_0 in an active term position, we cannot do this exactly with the current inference rule `Ind`. To solve this, we propose a more general induction inference rule, called *multi-clause induction* that allows us to induct on more than one clause at once:

$$\frac{formula \rightarrow \forall x_1, ..., x_n.F[x_1]...[x_n] \qquad L_1[\sigma_1]...[\sigma_n] \vee C_1 \qquad ... \qquad L_m[\sigma_1]...[\sigma_n] \vee C_m}{\texttt{cnf}(\neg formula \vee C_1 \vee ... \vee C_m)} \ (\texttt{IndMC})$$

where $L_1[\sigma_1]...[\sigma_n], ..., L_m[\sigma_1]...[\sigma_n]$ are ground literals, $F := \overline{L_1}[\sigma_1]...[\sigma_n] \vee ... \vee \overline{L_m}[\sigma_1]...[\sigma_n]$, $formula \rightarrow \forall x_1, ..., x_n.F[x_1]...[x_n]$ is a valid induction formula on n induction terms. So the proper way to find a proof for our previous example would be as follows:

1. Analysing the subgoal $\neg\texttt{even}(\texttt{add}(\sigma_0, \sigma_1))$, we find that the `even` term suggests an induction scheme with induction term σ_0.
2. We look for other clauses that have the same σ_0 induction term for some induction scheme and we find $\texttt{even}(\sigma_0)$.
3. The two induction schemes are combined, an induction formula is generated where each case contains the opposite sign literals for $\neg\texttt{even}(\texttt{add}(\sigma_0, \sigma_1))$ and $\texttt{even}(\sigma_0)$ disjuncted, then this induction formula is resolved with the two original clauses.

The induction formula we get is the following:

$$\begin{pmatrix} (\neg\texttt{even}(\texttt{0}) \vee \texttt{even}(\texttt{add}(\texttt{0}, \sigma_1))) \wedge (\neg\texttt{even}(\texttt{s}(\texttt{0})) \vee \texttt{even}(\texttt{add}(\texttt{s}(\texttt{0}), \sigma_1))) \wedge \\ \forall x.((\neg\texttt{even}(x) \vee \texttt{even}(\texttt{add}(x, \sigma_1))) \rightarrow (\neg\texttt{even}(\texttt{s}(\texttt{s}(x))) \vee \texttt{even}(\texttt{add}(\texttt{s}(\texttt{s}(x)), \sigma_1)))) \end{pmatrix} \\ \rightarrow \forall z.(\neg\texttt{even}(z) \vee \texttt{even}(\texttt{add}(z, \sigma_1)))$$

This gives after binary resolution with the two clauses and clausification, a large set of clauses from which only the following three is enough to get a refutation:

$$\left\{\begin{array}{llll} \{\neg\mathtt{even}(\mathtt{add}(0,\sigma_1)), & \mathtt{even}(\mathtt{s}(\mathtt{s}(\sigma_2))), & \mathtt{even}(\mathtt{s}(0))\}, & \\ \{\neg\mathtt{even}(\mathtt{add}(0,\sigma_1)), & \neg\mathtt{even}(\mathtt{add}(\mathtt{s}(\mathtt{s}(\sigma_2)),\sigma_1)), & \mathtt{even}(\mathtt{s}(0))\}, & \\ \{\neg\mathtt{even}(\mathtt{add}(0,\sigma_1)), & \mathtt{even}(\mathtt{add}(\sigma_2,\sigma_1)), & \mathtt{even}(\mathtt{s}(0)), & \neg\mathtt{even}(\sigma_2)\} \end{array}\right\}$$

After applying simplifications exhaustively, we get this new set:

$$\left\{\begin{array}{lll} \{\neg\mathtt{even}(\sigma_1), & \mathtt{even}(\sigma_2)\}, & \\ \{\neg\mathtt{even}(\sigma_1), & \neg\mathtt{even}(\mathtt{add}(\sigma_2,\sigma_1))\}, & \\ \{\neg\mathtt{even}(\sigma_1), & \mathtt{even}(\mathtt{add}(\sigma_2,\sigma_1)), & \neg\mathtt{even}(\sigma_2)\} \end{array}\right\}$$

Each clause can be binary resolved with $\mathtt{even}(\sigma_1)$:

$$\left\{\left\{\mathtt{even}(\sigma_2)\right\},\left\{\neg\mathtt{even}(\mathtt{add}(\sigma_2,\sigma_1))\right\},\left\{\mathtt{even}(\mathtt{add}(\sigma_2,\sigma_1)),\neg\mathtt{even}(\sigma_2)\right\}\right\}$$

In the end the three clauses can be binary resolved with each other pairwise and we get the empty clause.

Interestingly, this example can be proven without multi-clause induction using a different induction scheme that we get from combining the schemes suggested by `add` and `even` with strengthened induction hypotheses:

$$\left(\begin{array}{c} \mathtt{even}(\mathtt{add}(0,\sigma_1))\wedge \\ (\forall x_0.\mathtt{even}(\mathtt{add}(0,x_0))\rightarrow\mathtt{even}(\mathtt{add}(\mathtt{s}(0),\sigma_1)))\wedge \\ (\forall x,x_1,x_2.)\left(\begin{array}{c}(\mathtt{even}(\mathtt{add}(x,x_1))\wedge\mathtt{even}(\mathtt{add}(\mathtt{s}(x),x_2)))\rightarrow \\ \mathtt{even}(\mathtt{add}(\mathtt{s}(\mathtt{s}(x)),\sigma_1))\end{array}\right)\end{array}\right)\rightarrow\forall z.\mathtt{even}(\mathtt{add}(z,\sigma_1))$$

We hypothesize that one of the strengths of multi-clause induction is that when a conjecture can be only proven with multiple applications of induction – partly because the induction hypotheses cannot match the induction step consequent at first and using this rule, – all still usable induction hypotheses can be carried on for later use with the same set of Skolem constants. We demonstrate this with the following example:

Example 4.4.1 Prove the following conjecture:

$$\forall x.\mathtt{rev}(\mathtt{rev}(x))=x$$

where `rev` is the recursive function definition reversing a `list` argument:

$$\begin{array}{l}\textbf{fun}\ \mathtt{rev}(x:\mathtt{list})\rightarrow\mathtt{list}:= \\ \quad\textbf{match}\ x\ \textbf{with} \\ \quad\quad(\mathtt{nil}\rightarrow\mathtt{nil}) \\ \quad\quad(\mathtt{cons}(y,z)\rightarrow\mathtt{app}(\mathtt{rev}(z),\mathtt{cons}(y,\mathtt{nil})))\end{array}$$

First, negate and Skolemize the conjecture:

$$\mathtt{rev}(\mathtt{rev}(\sigma_0)) \neq \sigma_0$$

The first induction is rather straightforward since we can only induct on σ_0. We get the following set of clauses:

$$\left\{\begin{array}{ll}\{ & \mathtt{rev}(\mathtt{rev}(\mathtt{nil})) \neq \mathtt{nil}, \quad \mathtt{rev}(\mathtt{rev}(\sigma_2)) = \sigma_2 \quad \}, \\ \{ & \mathtt{rev}(\mathtt{rev}(\mathtt{nil})) \neq \mathtt{nil}, \quad \mathtt{rev}(\mathtt{rev}(\mathtt{cons}(\sigma_1, \sigma_2))) \neq \mathtt{cons}(\sigma_1, \sigma_2) \quad \}\end{array}\right\}$$

After simplification, the base case can be removed as it is $\mathtt{nil} \neq \mathtt{nil}$ and we get:

$$\{\{\mathtt{rev}(\mathtt{rev}(\sigma_2)) = \sigma_2\}, \{\mathtt{rev}(\mathtt{app}(\mathtt{rev}(\sigma_2), \mathtt{cons}(\sigma_1, \mathtt{nil})) \neq \mathtt{cons}(\sigma_1, \sigma_2))\}\}$$

Unfortunately, the induction hypothesis cannot be used: there is no $\mathtt{rev}(\mathtt{rev}(\sigma_2))$ subterm in the inequality and no simplification ordering allows us to rewrite σ_2 into $\mathtt{rev}(\mathtt{rev}(\sigma_2))$ either. Another option would be to induct on σ_2 but then the new induction step case would be unprovable (as it is no longer true). We instead use multi-clause induction, inducting on $\mathtt{rev}(\sigma_2)$ from both clauses. This is possible if we extend Heuristic 3.12.1 to also count occurrences of complex terms in other premises. We get thus the following induction formula:

$$\left(\begin{array}{c}(\mathtt{rev}(\mathtt{nil}) \neq \sigma_2 \vee \mathtt{rev}(\mathtt{app}(\mathtt{nil}, \mathtt{cons}(\sigma_1, \mathtt{nil})) = \mathtt{cons}(\sigma_1, \sigma_2)))\wedge \\ (\forall y, z.)\left(\begin{array}{c}(\mathtt{rev}(z) \neq \sigma_2 \vee \mathtt{rev}(\mathtt{app}(z, \mathtt{cons}(\sigma_1, \mathtt{nil})) = \mathtt{cons}(\sigma_1, \sigma_2))) \rightarrow \\ (\mathtt{rev}(\mathtt{cons}(y, z)) \neq \sigma_2 \vee \mathtt{rev}(\mathtt{app}(\mathtt{cons}(y, z), \mathtt{cons}(\sigma_1, \mathtt{nil})) = \mathtt{cons}(\sigma_1, \sigma_2)))\end{array}\right) \\ \rightarrow \forall x.(\mathtt{rev}(x) \neq \sigma_2 \vee \mathtt{rev}(\mathtt{app}(x, \mathtt{cons}(\sigma_1, \mathtt{nil})) = \mathtt{cons}(\sigma_1, \sigma_2)))\end{array}\right)$$

After clausification, we only need 3 of the resulting 6 clauses to get a refutation:

$$\left\{\begin{array}{ll}\{ & \mathtt{cons}(\sigma_1, \sigma_2) \neq \mathtt{rev}(\mathtt{app}(\mathtt{cons}(\sigma_3, \sigma_4), \mathtt{cons}(\sigma_1, \mathtt{nil}))), \quad \mathtt{rev}(\mathtt{nil}) = \sigma_2\}, \\ \{ & \sigma_1 = \mathtt{rev}(\mathtt{cons}(\sigma_3, \sigma_4)), \quad \mathtt{rev}(\mathtt{nil}) = \sigma_2\}, \\ \{ & \mathtt{rev}(\sigma_4) \neq \sigma_1, \mathtt{cons}(\sigma_1, \sigma_2) = \mathtt{rev}(\mathtt{app}(\sigma_4, \mathtt{cons}(\sigma_1, \mathtt{nil}))), \quad \mathtt{rev}(\mathtt{nil}) = \sigma_2\}\end{array}\right\}$$

After simplification and an equality resolution in the last clause, we get:

$$\left\{\begin{array}{ll}\{ & \mathtt{cons}(\sigma_1, \sigma_2) \neq \mathtt{app}(\mathtt{rev}(\mathtt{app}(\sigma_4, \mathtt{cons}(\sigma_1, \mathtt{nil}))), \mathtt{cons}(\sigma_3, \mathtt{nil})), \quad \mathtt{nil} = \sigma_2\}, \\ \{ & \sigma_1 = \mathtt{app}(\mathtt{rev}(\sigma_4), \mathtt{cons}(\sigma_3, \mathtt{nil})), \quad \mathtt{nil} = \sigma_2\}, \\ \{ & \mathtt{cons}(\sigma_1, \mathtt{rev}(\sigma_4)) = \mathtt{rev}(\mathtt{app}(\sigma_4, \mathtt{cons}(\sigma_1, \mathtt{nil}))), \quad \mathtt{nil} = \sigma_2\}\end{array}\right\}$$

Now the first literal in the first clause can be rewritten with the first equality from the last clause via superposition:

$$\left\{\begin{array}{ll}\{ & \mathtt{cons}(\sigma_1, \sigma_2) \neq \mathtt{app}(\mathtt{cons}(\sigma_1, \mathtt{rev}(\sigma_4)), \mathtt{cons}(\sigma_3, \mathtt{nil})), \quad \mathtt{nil} = \sigma_2\}, \\ \{ & \sigma_1 = \mathtt{app}(\mathtt{rev}(\sigma_4), \mathtt{cons}(\sigma_3, \mathtt{nil})), \quad \mathtt{nil} = \sigma_2\}\end{array}\right\}$$

Simplifying this, we get:

$$\left\{\begin{array}{ll}\{ & \mathtt{cons}(\sigma_1,\sigma_2) \neq \mathtt{cons}(\sigma_1,\mathtt{app}(\mathtt{rev}(\sigma_4),\mathtt{cons}(\sigma_3,\mathtt{nil}))),\quad \mathtt{nil}=\sigma_2\},\\ \{ & \sigma_1 = \mathtt{app}(\mathtt{rev}(\sigma_4),\mathtt{cons}(\sigma_3,\mathtt{nil})),\quad \mathtt{nil}=\sigma_2\}\end{array}\right\}$$

Using the term algebras injectivity, the first inequality can be expanded into two inequalities:

$$\left\{\begin{array}{ll}\{ & \sigma_1 \neq \sigma_1, \sigma_2 \neq \mathtt{app}(\mathtt{rev}(\sigma_4),\mathtt{cons}(\sigma_3,\mathtt{nil})),\quad \mathtt{nil}=\sigma_2\},\\ \{ & \sigma_1 = \mathtt{app}(\mathtt{rev}(\sigma_4),\mathtt{cons}(\sigma_3,\mathtt{nil})),\quad \mathtt{nil}=\sigma_2\}\end{array}\right\}$$

After a trivial inequality removal in the first clause, we can binary resolve the two to get simply $\{\{\mathtt{nil}=\sigma_2\}\}$ which can be superposed $\{\mathtt{rev}(\mathtt{app}(\mathtt{rev}(\sigma_2),\mathtt{cons}(\sigma_1,\mathtt{nil})) \neq \mathtt{cons}(\sigma_1,\sigma_2))\}$ from the result clauses of the first induction to get the empty clause.

Using strengthening of induction hypothesis makes the application of the multi-clause induction rule harder, since some of the induction hypotheses will be non-ground. Nevertheless, we find both of these techniques very useful. We conclude this section by proving soundness of `IndMC`:

Theorem 4.4.1 The inference rule `IndMC` is sound.

Proof. We use the same assumptions as in the soundness proof of `Ind`, that is, we have a valid induction formula as a left premise. After clausification, there may be clauses that do not contain any literals from the induction main conclusion – this can be a result of added definitions during clausification, however if one contains any literal from the main conclusion, it contains all literals unmodified since the conclusion was not negated. After the first binary resolution of any such clause with one of the premises, we get a substitution bounding some of the variables in the resulting clause. We have a bijective correspondence between these variables and the induction terms by construction, hence all subsequent binary resolutions can be performed with all other premises. From the soundness of binary resolution, this means that the resolvents are correctly derived and the inference rule is sound. □

4.5 Clause selection for multi-clause induction

As opposed to other inference rules such as superposition or binary resolution, multi-clause induction has an arbitrary number of premises – it is essential therefore to properly reduce the number of clauses we want to use together as premises. We will now develop a heuristic to do so.

First, as with single-clause induction, we usually select a negative literal from a clause as an inductive goal stemming from the original goal which we would like to refute, the clause of this literal is the *main premise* of the inference. Then, we add a number of

side premises which we suspect can be relevant to the proof of the main premise. We cannot just add any side premise since that would blow up the search space, so we select clauses that contain literals which are structurally similar. Specifically, as we are dealing with ground literals from the main premises, we can just select other literals with the same Skolem constants and function/predicate symbols. Note that given a clause, we may induct on several of its literals separately and use other non-unit clauses as side premises in the same manner.

What we are aiming at is that eventually we can use these side premises to simplify or get rid of the main premise. This can happen in several ways: for example, if the main premise is a disequality, after simplifying the inductive step conclusion or base case corresponding to it, we can rewrite it or binary resolve it with an equality. If it is a non-equality literal, we also have the chance to rewrite it with an equality and eventually binary resolve it with a positive literal. In either case, we use positive literals for a negative subgoal.

Another good reduction would be to only include side premises that have similar terms as the main premise, e.g. with function symbols and constants more or less in the same positions. This "more or less" is unfortunately very vague and the problem of matching subterms in general is NP-hard. Therefore, we only use cheap heuristics without having to analyze the structural nuances of the literals: we select literals that have any subset of Skolem constants that of the main premise. If a side premise contains a Skolem constant that the main premise does not, they probably will not match later due to this constant, therefore it is often useless to induct on them together. On the other hand, if this constant pops up in the main premise later because of some rewriting or the side premise is simplified and the constant is removed, after that we still have the chance to use them together in induction. Our final heuristic is the following:

Heuristic 4.5.1 (Side premise selection for `IndMC`) For a negative literal with set of Skolem constants S, we select all positive literals with sets of Skolem constants $T \subseteq S$.

CHAPTER 5

Implementation

We have implemented the techniques for inductive reasoning in the state-of-the-art theorem prover Vampire [KV13]. Our implementation and extensions in Vampire required ∼3800 lines of code. In this chapter we look at how this was done in detail.

5.1 Parsing

There are currently 3 main input formats supported by theorem provers/SMT solvers: TPTP [Sut17], and SMT-LIB version 1 and 2 [BFT16]. Of the three formats, only version 2 of SMT-LIB supports function definitions and the `match` keyword. First, we look at how these are parsed and processed, then we show some methods to detect the same function definition structures when we are dealing with an input without these new keywords.

Before converting a formula to a first-order normal form, it is allowed to contain special terms, such as `bool` expressions in equalities, `ite` or `let` expressions and so on. We introduced a special symbol for `match` which contains a variable as its first argument and then $2n$ arguments for the n pairs containing the constructs (s_i, t_i) described in Section 2.1. As per the SMT-LIB standard, we allow only new variables in the matched expressions s_i. Also, exhaustiveness and mutual exclusiveness are checked with an addition that the last matched expression can be a single variable matching all remaining expressions (i.e. an *else* branch).

We also added the not-yet-implemented `define-fun-rec` keyword. This is very similar to the already existing `define-fun` implementation, except that in the latter one cannot use terms with the function that is defined. Furthermore, recursive functions as well as non-recursive ones are marked as a function definitions.

5.2 Preprocessing

In first-order theorem proving and in Vampire in particular, there are multiple stages of preprocessing after the input has been parsed and before the saturation begins. These include eliminating the special expressions like `ite`, `let` and `match` expressions, converting the formulas to negation normal form and so on. We introduced an additional step to get extra information used by the induction inference and its heuristics before the preprocessing is done on these expressions, that is, when the information is not yet scattered into multiple formulas or mixed with other axioms making it harder to collect.

5.2.1 Conversion to normal form

Vampire has a preprocessing pipeline consisting of many different steps based on the options supported to the prover. We only deal with the part that eliminates special symbols and converts formulas to clausal normal forms. There are two main methods for that, one is called `FOOL2FOL` [KKV15], the other `VCNF` [RSV16, KKSV16].

We first extend the `FOOL2FOL` conversion with the new keyword `match` as follows. This algorithm maintains a set of unprocessed formulas that cannot be directly converted to normal form yet. It also introduces new symbols if it finds some complex term that needs to be eliminated. In any unprocessed formula, all such terms are eliminated in a single round. For each expression $\texttt{match}(x, s_1, t_1, ..., s_n, t_n)$ with free variables $y_1, ..., y_m$ in one such unprocessed formula, if the expression has a non-`bool` sort, we add a new function symbol g of arity m, otherwise a new predicate symbol q of arity m. Then, for each $1 \leq i \leq n$ we add the following formula to the set of unprocessed formulas in case of a non-`bool` sort:

$$x = s_i \rightarrow g(y_1, ..., y_m) = t_i$$

For a `bool` result sort, we add this formula instead:

$$x = s_i \rightarrow q(y_1, ..., y_m) \leftrightarrow t_i$$

In the end, we replace the original occurrence of the `match` expression with $g(y_1, ..., y_m)$ or $q(y_1, ..., y_m)$, respectively. We do this for other terms that need to be eliminated from the original formula, convert it to clausal normal form and proceed with the remaining unprocessed formulas.

The other method `VCNF` covers slightly different stages of the preprocessing pipeline but it nevertheless also eliminates complex terms such as `ite` and `let` expressions, so we extended this algorithm as well. In a nutshell, the `VCNF` conversion processes a set of sets of formulas until they all contain only literals and therefore the result is in clausal normal form. During this process, when we encounter occurrences of a `match` expression in the current formula F in a set, i.e.

$$\{..., \{..., F[\texttt{match}(x, s_1, t_1, ..., s_n, t_n)], ...\}, ...\}$$

we replace this set with n duplicates:

$$\{..., \{..., x \neq s_1, F[t_1], ...\}, ..., \{..., x \neq s_n, F[t_n], ...\}, ...\}$$

The `VCNF` method allows us to easily pass the information that an equality is marked a function definition from the original formula to the resulting clauses, as opposed to `FOOL2FOL` where the newly introduced symbols make it hard to mark the original $\mathtt{f}(\overline{x}) = t$ since it might be in the form $\mathtt{f}(\overline{x}) = g(\overline{y}) \vee C$ and $g(\overline{y}) = t \vee D$, present in two clauses separately. Although this can be done by using e.g. symbol elimination, we omitted such additional steps and used only `VCNF` for function definition rewriting.

5.2.2 Processing function definitions

We implemented a preprocessor which processes all input formulas before they are converted to normal forms. When it encounters the result of a `define-fun` or `define-fun-rec` block, i.e. a formula marked as function definition, it creates the r-descriptions the block contains and adds a corresponding induction template to the set of templates. It also maintains a set of partial induction templates based on guessing whether an axiom not marked as a function definition contains a function definition or not. When such possible function definitions are found, it tries to discover recursive calls in them. In the end, all induction templates are checked for well-foundedness by finding an ordered separation if it exists. If this check fails, the induction template is discarded as it may produce invalid induction formulas, rendering the induction inference rules unsound.

The algorithm in Figure 5.1 describes how the r-descriptions of a function definition are collected. It parses the branching given by `ite` and `match` expressions, making substitutions where possible then processes each branch term with `processCase` (Figure 5.2). The helper function `processCase` adds any recursive calls found in the current branch to the set of recursive calls. When a marked function definition $f(\overline{args}) = body$ is found, the call $\mathtt{processBody}(body, f(\overline{args}), \emptyset)$ results in the r-descriptions of that function definition.

5.2.3 Function definition discovery

Any other axioms from the input not marked as function definitions are processed as possible function definitional axioms by looking at specific patterns shown in Figure 5.3. We only look at certain formulas here, because otherwise we would have too many possibilities e.g. in a disjunction with n disjuncts, where we would have at least 2^n possibilities to look at, not even counting complex subformulas. Existential formulas are also excluded and for negations, only the direct subformula is considered, i.e. `bool` function base cases with false values.

The `isHeader` function (Figure 5.4) takes only function headers with constructor terms or variables as arguments. It may be the case that a recursive call cannot be discovered directly because it is hidden inside another function definition, for instance we only discover the equality $f(t_1, ..., t_n) = g(t_1, ..., t_n)$ and there is another equality $g(s_1, ..., s_n) =$

Algorithm 5.1: processBody

Input: $t \in \mathcal{T}, f(\overline{args}) \in \mathcal{T}$ function header, c set of conditions

```
1  if t is variable then
2  |  return (h, ∅, c);
3  else if t is match(s, s_1, t_1, ..., s_n, t_n) then
4  |  res ← ∅;
5  |  for 1 ≤ i ≤ n do
6  |  |  res ← res ∪ processBody(t_i, f(args)[s/s_i], c);
7  |  end
8  |  return res;
9  else if t is ite(p, t_1, t_2) then
10 |  return processBody(t_1, f(args), c ∪ {p}) ∪
   |   processBody(t_2, f(args), c ∪ {¬p});
11 else if t is let(x_1, t_1, ..., x_n, t_n, t) then
12 |  return processBody(t[x_1/t_1]...[x_n/t_n], f(args), c)
13 else
14 |  return (f(args), processCase(t, f), c)
15 end
```

Figure 5.1: Algorithm `processBody`

$f(s_1, ..., s_n)$ with the recursive call. Although we may have to go down arbitrarily deep into these definitions, the lack of a recursive call does not make an induction formula invalid, the only issue is that a probably necessary induction hypothesis will be missing. Therefore, we choose not to check such indirect recursive calls.

Expressions of the form $t_1 = t_2$ and $F_1 \leftrightarrow F_2$ must be checked both directions and even then, we may not be able to tell which direction is the intended. For example, an equality $f(\mathtt{s}(x), y) = f(x, \mathtt{s}(y))$ is surely a recursive definition but by itself, no orientation can be deduced. Another example is $f(\mathtt{nil}, y) = g(y)$, where both sides of the equality are potential function headers. To find out which orientation is the best, we maintain a set of sets of possible induction templates. When we have such an ambiguous equality, we duplicate each set of induction templates in the maintained set and add one orientation to the first subset, the other to the second. This way, in the end we get all possible induction template combinations that we could detect.

After processing all input formulas, we can look at the possible induction template configurations and find out which ones are not well-suited for our purposes. In particular, we first do a well-foundedness and well-definedness check on each induction template in a

Algorithm 5.2: processCase

Input: $t \in \mathcal{T}, f \in \mathcal{F}$

```
1  res ← ∅;
2  if t is f(t_1, ..., t_n) then
3  |   res ← res ∪ t;
4  |   for 1 ≤ i ≤ n do
5  |   |   res ← res ∪ processCase(t_i, f);
6  |   end
7  else if t is g(t_1, ..., t_m) ∧ f ≠ g then
8  |   for 1 ≤ i ≤ m do
9  |   |   res ← res ∪ processCase(t_i, f);
10 |   end
11 return res
```

Figure 5.2: Algorithm `processCase`

set. As it might be that there is no set which contains only well-founded and well-defined templates, we have to select which one is the best. For each set containing k templates that are not well-founded and l that are not well-defined, we calculate a score as follows:

$$score = l + 5k$$

We prefer well-founded relations over well-defined ones, therefore the violation of the former is penalized more. We then select the set with the least score. If the selected set has score greater than 0, we have to "fix" it. If a template does not give a well-founded order on its active arguments, we simply discard it as it may give rise to unsound induction inferences. The template can also be over-defined or under-defined, i.e. some of the cases in its case distinction of arguments are matched by multiple function headers or are not matched by any function header, respectively. We make however a distinction here: while over-definedness can lead to e.g. non-confluent rewriting, it does not lead to unsoundness. On the other hand, generating induction schemes from a template that has incomplete case distinction can do so and we fix it by adding the missing cases. This is in fact done the same way as generating the base cases during combining two induction schemes. In the end, we mark all equalities to be used as function definitions by function definition rewriting and fix their orientation. After this, we can start the saturation. We demonstrate this with an example.

Algorithm 5.3: findFunctionDefinitions

Input: F formula, c set of formulas

```
1  if F is t1 = t2 then
2      if isHeader(t1) then
3          r1 ← processBody(t2, t1, c);
4      if isHeader(t2) then
5          r2 ← processBody(t1, t2, c);
6      if isWellFounded(r1) ∧ isWellFounded(r2) then
7          defs[f] ← (defs[f] × r1) ∪ (defs[f] × r2);
8      else if isWellFounded(r1) then
9          defs[f] ← defs[f] × r1;
10     else if isWellFounded(r2) then
11         defs[f] ← defs[f] × r2;
12 else if F is F is F1 ↔ F2 then
13     if F1 is p(args) ∧ isHeader(F1) then
14         r1 ← processBody(F2, F1, c)
15     if F2 is p(args) ∧ isHeader(F2) then
16         r2 ← processBody(F1, F2, c)
17     if isWellFounded(r1) ∧ isWellFounded(r2) then
18         defs[p] ← (defs[p] × r1) ∪ (defs[p] × r2)
19     else if isWellFounded(r1) then
20         defs[p] ← defs[p] × r1
21     else if isWellFounded(r2) then
22         defs[p] ← defs[p] × r2
23 else if F is p(args) ∨ F is ¬p(args) then
24     defs[p] ← defs[p] × (p(args), ∅, c)
25 else if F is F1 ∧ ... ∧ Fn then
26     for 1 ≤ i ≤ n do
27         findFunctionDefinitions(Fi, c)
28     end
29 else if F is F1 → F2 then
30     findFunctionDefinitions(F2, c ∪ F1)
```

Figure 5.3: Algorithm `findFunctionDefinitions`

Example 5.2.1 Given a set of equalities on functions `f` and `g`

$$\left\{\begin{array}{c} \mathtt{f}(\mathtt{cons}(x,y),z) = \mathtt{f}(y,\mathtt{cons}(x,z)), \\ \mathtt{f}(\mathtt{nil},x) = x, \\ \mathtt{g}(x) = \mathtt{f}(x,\mathtt{nil}) \end{array}\right\}$$

determine the orientation for each equality and find well-founded (and possibly well-defined) induction templates for `f` and `g`.

The second equality can be easily oriented, but the first and third can be oriented either ways. This gives us four possible scenarios for the two induction templates:

1. $\left\{\begin{array}{c} (\mathtt{f}(\mathtt{cons}(x,y),z),\{\mathtt{f}(y,\mathtt{cons}(x,z))\},\emptyset), \\ (\mathtt{f}(\mathtt{nil},x),\emptyset,\emptyset),(\mathtt{f}(x,\mathtt{nil}),\emptyset,\emptyset) \end{array}\right\}$ for `f` and $\emptyset$ for `g`
2. $\left\{\begin{array}{c} (\mathtt{f}(y,\mathtt{cons}(x,z)),\{\mathtt{f}(\mathtt{cons}(x,y),z)\},\emptyset), \\ (\mathtt{f}(\mathtt{nil},x),\emptyset,\emptyset),(\mathtt{f}(x,\mathtt{nil}),\emptyset,\emptyset) \end{array}\right\}$ for `f` and $\emptyset$ for `g`
3. $\left\{\begin{array}{c} (\mathtt{f}(\mathtt{cons}(x,y),z),\{\mathtt{f}(y,\mathtt{cons}(x,z))\},\emptyset), \\ (\mathtt{f}(\mathtt{nil},x),\emptyset,\emptyset) \end{array}\right\}$ for `f` and $\{(\mathtt{g}(x),\emptyset,\emptyset)\}$ for `g`
4. $\left\{\begin{array}{c} (\mathtt{f}(y,\mathtt{cons}(x,z)),\{\mathtt{f}(\mathtt{cons}(x,y),z)\},\emptyset), \\ (\mathtt{f}(\mathtt{nil},x),\emptyset,\emptyset) \end{array}\right\}$ for `f` and $\{(\mathtt{g}(x),\emptyset,\emptyset)\}$ for `g`

All four templates of `f` give a well-founded order but while the first and second are over-defined because the second argument is matched by both `nil` and a variable, the fourth is under-defined because it does not match a `cons` term in the first argument. Therefore the scores are 1, 1, 0 and 1, respectively, so we select the third which gives the same orientation as in the beginning of the example. We mark the first two equalities as function definitions of `f` and the last as function definition for `g` and fix their orientation.

5.2.4 Checking for well-foundedness

We now turn our attention to the well-foundedness check. For this, we need all active argument positions and all possible function header - recursive call pairs. The algorithm `checkWellFoundedness` (Figure 5.5) has two high-level parts: the first one calls the helper function `separateIntoSets` which maintains a set initially containing the set of all active argument position indices and separates this into non-empty sets which "change together" in all given recursive calls and thus candidates for being the first in the currently created lexicographic order.

The second part takes all the created sets of indices and tries to recursively extend them to a complete lexicographic order on the remaining indices with the remaining recursive calls, i.e. the ones in which none of the selected indices are changing – these correspond

Algorithm 5.4: isHeader

Input: $t \in \mathcal{T}$

```
1 if t is f(t_1, ..., t_n) then
2 |   for 1 ≤ i ≤ n do
3 |   |   if not isTermAlgebraCons(t_i) then
4 |   |   |   return false;
5 |   end
6 |   return true;
7 return false;
```

Algorithm 5.5: isTermAlgebraCons

Input: $t \in \mathcal{T}$

```
1 if t is variable then
2 |   return true;
3 else if t is c(t_1, ..., t_n) then
4 |   for 1 ≤ i ≤ n do
5 |   |   if not isTermAlgebraCons(t_i) then
6 |   |   |   return false;
7 |   end
8 |   return true;
9 return false;
```

Figure 5.4: Algorithms `isHeader` and `isTermAlgebraCons`

to cases we still need to create a lexicographic order for. The base of the recursion is when either we are out of recursive calls or out of indices. The result ordered separation is expanded with a new set of indices s by calling `addToOrderedSeparation(s)` whenever the recursive call returns true on line 9 of `checkWellFoundedness`.

Notice that we can select a set of indices which do not change in any of the recursive calls. This means that the we recurse on all pairs from R but this is not a problem since these indices then vacuously satisfy the condition in the lexicographic order definition.

We also implemented a similar well-foundedness check for induction schemes. This is invoked after a scheme is generated from a template and after two schemes are combined. While the first one detects implementation errors as all induction schemes at that point should be well-founded, the second is necessary to detect non-well-founded unions to discard them.

Algorithm 5.6: checkWellFoundedness

Input: I active argument position set, R recursive call - header pairs

```
1  if R is ∅ then return true;
2  if I is ∅ then return false;
3  S ← separateIntoSets(I, R);
4  for s ∈ S do
5  |  R' ← ∅;
6  |  for (f(s_1, ..., s_n), f(t_1, ..., t_n)) ∈ R do
7  |  |  if for any i ∈ s, s_i is not t_i then R' ← s;
8  |  end
9  |  if checkWellFoundedness(I \ s, R') then
10 |  |  addToOrderedSeparation(s);
11 |  |  return true;
12 end
13 return false;
```

Algorithm 5.7: separateIntoSets

Input: I active argument position set, R recursive call - header pairs

```
1  S ← {R};
2  for (f(s_1, ..., s_n), f(t_1, ..., t_n)) ∈ R do
3  |  fixed ← ∅;
4  |  subterm ← ∅;
5  |  for 1 ≤ i ≤ n do
6  |  |  if s_i is t_i then fixed ← fixed ∪ {i};
7  |  |  else if s_i is subterm of t_i then subterm ← subterm ∪ {i};
8  |  end
9  |  S' ← ∅;
10 |  for s ∈ S do
11 |  |  if s ∩ fixed is not ∅ then S' ← S' ∪ {s ∩ fixed};
12 |  |  if s ∩ subterm is not ∅ then S' ← S' ∪ {s ∩ subterm};
13 |  end
14 |  S ← S';
15 end
16 return S;
```

Figure 5.5: Algorithms `checkWellFoundedness` and `separateIntoSets`

5.2.5 Checking for well-definedness

The well-definedness check is very similar to the generation of base cases from Section 3.8. We maintain a set of available term mappings corresponding to all possible argument tuples for a function, from which we exclude the cases of the corresponding induction template. In the end the function is under-defined if the normalizations of the available term mappings are not empty.

For the over-definedness, we check that for each argument in each r-description, some value is excluded from one of the available term mappings. If no value is excluded, then it is a clear sign that some argument tuples are covered by more than one function branches. Also, since we cannot properly exclude an r-description if it has side conditions (that would require checking formula subsumption and validity which is not practically achievable for arbitrary nested formulas), for branches with side conditions we do not report non-well-definedness.

5.2.6 Generating and maintaining induction schemes

We mainly use the definitions for generating induction schemes from induction templates and function terms, however we would like to highlight some of the differences in the implementation. The first is that (as already mentioned) a single position in a function term can give multiple induction terms. This can be because we are inducting on complex terms or because the subterms themselves have multiple active positions. We use all such possible selected term tuples from a single induction term to generate induction schemes.

Another aspect worth mentioning is that during the generation of induction formulas, the number of resulting clauses and the size of those clauses are highly dependant on the induction schemes. For instance, after combining two induction schemes, each intersection can contain duplicate recursive call substitutions or ones that contain one another. We therefore discard those at the end of intersection creation in the following way: depending on whether we create strengthened induction hypotheses or not, given two recursive substitution $\{\sigma_0 \mapsto x, \sigma_1 \mapsto y\}$ and $\{\sigma_0 \mapsto x\}$, the first can contain the second or vice versa. This is because although in general the relation corresponding to the second always contains the first, this is only true if we can strengthen with the unused term σ_1 in the second. Otherwise, there will be a σ_1 in the induction hypothesis that cannot be matched with anything from the conclusion (since σ_1 is replaced in the conclusion with some variable and then Skolemized).

After cleaning up the recursive cases, we check for containment between the recursive r-description instances themselves and discard the ones that are contained by some other in the scheme. Moreover, we maintain a mutually exclusive set of r-description instances during the generation of base cases, since in principle there can be duplicates among those as well.

5.3 Inference rules

Most saturation algorithms maintain a set of active clauses from which they select the next clause to saturate (the given clause) and after applying appropriate inferences exhaustively on it, they remove it from the active clauses. This execution order means after saturating the given clause, it can no longer participate in any inference, so it has to be tried in all possible positions as a premise in all inference rules so that no inference with it is "missed". In particular, when using asymmetric inference rules with multiple premises, e.g. with superposition, we have to try the given clause as a *rewriting clause* as well as the *rewritten clause*. The former is usually called the *backward* direction, the latter the *forward* direction of the inference rule.

Another issue is to select other premises for an inference with a given clause. Naively searching through all active clauses would be a bottleneck, so saturation-based theorem provers usually use some kind of term/literal indexing structure to look up other clauses with certain structural parameters. Different inference rules require different special indices such as ones for complementary literals for binary resolution or terms that unify with an equality left-hand side for superposition. For a more detailed discussion on term indexing, see [RV01]. We now take a look at the inference rules we introduced in Chapter 4 with these issues in mind.

5.3.1 Multi-clause induction

In Chapter 4, we have already elaborated on how to use the inference rule `IndMC` in practice. We implemented a "forward" and a "backward" direction of `IndMC`, that is, one where the given clause is the main premise and one where it is the side premise. These cases do not overlap as we only select negative literals for main premises and positive literals for side premises. The index we used for the lookup is a *demodulation subterm index*, as it can be queried for certain subterms on both sides of equalities. We can simply query this index for instances of the induction terms which means exact subterms since all premises are ground.

The forward direction is rather straightforward: we have a negative main premise, for which we select all possible positive side premises. Unfortunately, the other direction is a bit more problematic: to each side premise corresponds a set of main premises. To imitate the forward direction, we have to select more side premises *based on the main premise, not on the side premise*. This, however, would mean that as we saturate side premises corresponding to a particular main premise, in each turn we try to prove the main premise with a different set of side premises. This can really affect the proof search in a negative way. Another possibility would be to track which sets of side premises were inducted on with which main premises in different timepoints of the saturation. This would waste a lot of resources, so we refrain from doing so.

Instead, for each side premise we only select main premises. Even though inducting on all such main premises with the side premise in a single induction formula is valid and can be successful, due to the disjunctions in negations, the resulting clauses will be much

bigger without gaining anything. Although multiple side premises may be necessary for a successful induction on a single main premise in the forward direction, multiple main premises do not help in proving each other with a single side premise, hence it makes sense to induct on all those side-main premise pairs separately in the backward direction.

Summarizing the above, if we find a negative literal during saturation, we select all positive literals with Skolem constants also present in the negative literal and induct on all of them in a single inference. Otherwise, we select all negative literals that contain a superset of Skolem constants that of the current literal and induct on them one-by-one with the current literal as side premise.

5.3.2 Function definition rewriting

We talked about two inference rules for expanding function headers into their definitions in Chapter 4: one is the demodulation rule where the ordering and the function orientation coincides, the other is a paramodulation where the function orientation differs from the one given by the ordering.

The former case is relatively simple to implement: we only have to change how the *demodulation LHS index*, the index which stores the left-hand sides of unit equalities, handles the unit function definitions. When we provide this index with a new clause containing a single function definition equality literal, we check whether the left-hand side of the equality w.r.t. the ordering is the same as the function header – if that happens, the function header is added to the index, otherwise it is not. This index is then used in *forward demodulation* where we simplify the given clause with other active unit clauses. For *backward demodulation* where we simplify other clauses with the given clause, we have to modify the inference rule itself such that only those function definitions are used where the ordering coincides with the orientation.

For the latter case, we implemented a new inference rule and a new index. The inference rule has a backward and a forward direction but the two directions never overlap. Specifically, any function definition as given clause is used to rewrite other clauses (backward direction) but it is not rewritten by other clauses – we prohibited such inferences. On the other hand, a non-function definition as given clause is only rewritten by other active function definitional clauses (forward direction) and they do not rewrite anything themselves in this rule. For the forward direction, an index was added which stores only function definition headers – one per each clause containing a function definition –, so we can query what subterms of the given clause can be rewritten by these – in fact, we query generalizations of any subterm from the index (as opposed to unifications when using superposition). The forward direction uses the demodulation subterm index as described also for `IndMC` and instances of any function header are queried from this index.

CHAPTER 6

Results

We now turn our attention to experimenting with the techniques we have described so far. First we look at what new options has been added to Vampire to enable the new features, then we take a look at the benchmarks, what combinations of techniques were used to solve these and the analysis of the results.

6.1 Options

After implementing the techniques described in Chapters 3–4 in Vampire, we also added a set of flags that enable these when needed. There were previously three variants named `one`, `two` and `three` for structural induction that can be enabled with the flag `--structural_induction_kind` (or `-sik` for short). Even though the induction formula generation we described in Chapter 3 is similar to `one`, we would like to measure them separately, therefore we added a new variant `four`.

The Vampire options hierarchy allows us to add dependencies to each flag. We divided the features into three main sets: one contains all features closely related to induction, depending on structural induction kind `four`, the second contains only the function definition rewriting and the third contains the function definition discovery and analysis features, since these are somewhat unrelated things. Also, some induction features that are "cheap" to use and at the same time particularly useful during proof search are enabled directly with the variant `four`. These include the induction scheme subsumption and merging and induction term occurrence generalization.

The following flags are dependent on the variant `four`:

- `--induction_multiclause` (or `-indmc`) enables multi-clause induction.
- `--induction_strengthen` (or `-indstr`) enables strengthening the induction hypotheses.

- `--induction_term_occurrence_selection_heuristic` (or `-indtosh`) chooses the heuristic to select the term occurrences described in Section 3.11 with values `one` and `two`, respectively.
- `--simplify_before_induction` (or `-inds`) only allows induction on a literal if it contains no subterms simplifiable by any function definition.

There are flags that we reused since they are already implemented in some other induction variants:

- `--induction_unit_only` (or `-indu`) disallows inducting on literals in non-unit clauses. We extended this to multi-clause induction where it disallows selecting non-unit side-clauses.
- `--induction_neg_only` (or `-indn`) disallows inducting on positive literals.
- `--induction_on_complex_terms` (or `-indoct`) enables inducting on complex terms.

We also added one flag for enabling function definition rewriting, namely `--function_definition_rewriting` (or `-fnrw`).

6.2 Benchmarks and experimental setup

Since the success of induction depends heavily on having the correct function definitions, first we wanted to exclude any errors stemming from function definition discovery and use only function definition blocks and the `define-fun-rec` keyword.

Unfortunately, the SMT-LIB community still uses benchmarks almost exclusively without this keyword and other, relatively outdated and non-official benchmarks such as the TIP library [SJC15] use non-standard keywords that we cannot parse. Therefore, we used the benchmarks from [HHK+20] denoted by VIB from now on (which stands for Vampire Inductive Bencmarks). This set contains more than 3000 problems of increasing difficulty mostly for integer and list problems and also some for binary trees. We also extended the VIB set with some problems presented in this thesis. We categorized the benchmarks of VIB into sets, with some categories that are bigger and containing incrementally larger problems. Every problem except for the ones with `even` contain a single universally quantified literal. E.g. the `add` problems are categorized as follows:

- `add` assoc: associativity problems with 1 variable and occurrences of this variable ranging from 6 to 20.
- `add` mix: problems combining associativity and commutativity with number of variables ranging from 2 to 5 and overall occurrences of these ranging from 6 to 14.

- `add` comm: only commutativity with 2 variables and 2 occurrences.
- `add s 0` mix: problems combining associativity and commutativity with multiple variables and `0`s possibly inside `s` constructor terms, with occurrences ranging from 6 to 60.

After this, we looked at some of the inductive problem sets of SMT-COMP such as UFDTLIA that contain some well-known hard problems of inductive reasoning which required function definition discovery as well.

The experiments were conducted with the help of BenchExec which is especially suited for running theorem provers parallel on large sets of problems [BLW19, Bey16]. Each Vampire instance was provided one processor core, 16 GB of memory and a time limit of 300 seconds (enforced by both the Vampire flag `−t 300` and BenchExec). We used four versions of Vampire:

1. The first (denoted by Vampire) is from the master branch of Vampire as of January 2021 with options found working best described in [HHK+20], namely `−ind struct −indgen on −indoct on`.
2. The second (denoted by Vampire*) is from the feature branch containing the implementation of this thesis, where we manually disabled structural induction kind `four`, so that we can compare the master branch to this branch and we are also able to evaluate function definition rewriting independently. This runs with the same set of options as for Vampire if not stated otherwise.
3. The third (denoted by Vampire**) is from the feature branch of our implementation with default options `−ind struct −sik four`.
4. The fourth version (denoted by Vampire†) is a special one since it uses a combination of techniques via *portfolio mode*, see Section 6.6.

6.3 An overview

Before diving into the overall analysis of results given by the four Vampire variants on the benchmark sets, we take a quick look at what main problem types there are that are unsolvable for the original Vampire (at least with these options and within the given resource limits). These were specifically targeted by our thesis (some given as examples for the corresponding techniques they require) and were eventually proven by the most successful variant Vampire†. We also included some problems that are solvable with the original variant but cannot be solved with the new techniques – mainly due to a lack of proper generalizations over induction term occurrences. The comparison is shown in Figure 6.1 with `add` and `mul` shortened to + and ·, respectively for clarity.

Problem	Vampire	Vampire†
$\forall x, y.(y + (x + x)) + (x + y) = (x + (y + x)) + (x + y)$	✓	
$\forall x, y.(y + x) + (((y + y) + x) + x) = x + (y + (((y + y) + x) + x))$	✓	
$\forall x.\texttt{rev}(\texttt{rev}(x)) = x$		✓
$\forall x, y.(\texttt{even}(x) \wedge \texttt{even}(y)) \rightarrow \texttt{even}(x + y)$		✓
$\forall x, y, z.\texttt{fltn}(\texttt{node}(\texttt{node}(x, y, x), z, x)) = \texttt{fltn}(\texttt{node}(x, y, \texttt{node}(x, z, x)))$		✓
$\forall x.x \cdot (x \cdot x) = (x \cdot x) \cdot x$		✓
$\forall x, y, z.x \cdot (y + z) = (x \cdot y) + (x \cdot z)$		✓

Figure 6.1: Selected problems

6.4 Experiments with induction formula generation techniques

Our first set of benchmarks tested Vampire against Vampire** with and without other options that depend on the fourth variant, namely induction hypothesis strengthening (−`indstr on`), complex term induction (−`indoct on`) and multi-clause induction (−`indmc on`). The results are shown in Figure 6.2. The first table just lists the number of solved problems for each category in VIB and for UFDTLIA, color coded by the ratio between the number of solved problems and all problems in each category, so a darker blue shade is better.

This table shows that in case of VIB benchmarks Vampire** in itself is a great improvement over the previous version Vampire which mostly utilizes naive enumeration of generalizations over (complex) induction terms. We note however that while the VIB set contains some really large benchmarks which requires scaling of any proof search method, it lacks variety in terms of the recursive functions used, therefore, our induction term selection heuristics may be biased towards this set, while not particularly well-suited for other variations of recursive functions. This is reflected in the UFDTLIA benchmarks, where the difference between the two methods is only marginal.

Turning our attention to the other options, we can notice that all runs with different options enabled using Vampire** solved more or less the same number of problems. We can spot some clear differences between the categories, such as the two solved problems for `even` (which can be solved by these two options as discussed in Chapter 4), but in general it is hard to tell which one is better. To be able to differentiate between problems that are uniquely solved in a run among all runs shown in Figure 6.2, we created a second table, which shows exactly this number for each run.

The second table uncovers several interesting things: we can see for example that complex term induction solved the most problems uniquely, mostly in categories where we expect bad scaling due to a large number of variables, subterms and commutations between these. Also, induction hypothesis strengthening also turned out to be really well-suited for these

			Vampire	Vampire**			Options
					✓		−indoct on
						✓	−indstr on
							✓ −indmc on
Problems		Count	Solved problems				
list	app	313	72	238	252	243	232
	pref	696	94	687	642	693	688
	rev	4	2	2	2	2	2
nat	add assoc	312	81	261	252	275	245
	add mix	830	103	263	337	230	248
	add comm	2	2	2	2	2	2
	add s 0 mix	486	81	124	131	130	121
	mul	6	0	0	0	0	0
	even	2	0	0	0	1	1
	equal	4	2	3	2	3	3
	leq	696	117	696	680	696	696
	dup	1	0	0	0	0	0
btree	fltn	3	0	0	0	1	0
nat +list	add +app	1	1	1	1	1	1
UFDTLIA		327	144	149	149	166	151
All		3683	699	2426	2450	2443	2390

Problems		Count	Uniquely solved problems				
list	app	313	0	1	21	1	2
	pref	696	0	0	0	0	0
	rev	4	0	0	0	0	0
nat	add assoc	312	0	2	11	7	0
	add mix	830	4	6	64	11	2
	add comm	2	0	0	0	0	0
	add s 0 mix	486	0	1	5	7	1
	mul	6	0	0	0	0	0
	even	2	0	0	0	0	0
	equal	4	0	0	0	0	0
	leq	696	0	0	0	0	0
	dup	1	0	0	0	0	0
btree	fltn	3	0	0	0	1	0
nat +list	add +app	1	0	0	0	0	0
UFDTLIA		327	9	0	0	19	2
All		3683	13	10	101	46	7

Figure 6.2: Induction formula generation options compared

problems and also for some in the UFDTLIA set. Finally, the original implementation Vampire solved some problems uniquely, which shows the strengths of its generalization method over our simple heuristics. These differences between the measured techniques suggest that combining them would give even better results which we will do in the next sections.

6.5 Experiments with different preprocessing methods and function definition rewriting

As function definition rewriting is mostly independent of the induction formula generation, we wanted separate treatment for it in the experiments. Since this technique also depends on the method `VCNF` as described in Chapter 5, we needed to compare the new version to the old versions where this preprocessing method is used without function definition rewriting. Another difficulty was to see if the master branch and the feature branch containing the implementation of this thesis give the same results when run with the old method, hence we introduced the variant Vampire*.

The results are shown in Figure 6.3. This has the same structure as Figure 6.2, one table collecting the overall solved problems and one for uniquely solved problems. The first two columns compare Vampire and Vampire* with their default options. There are slight differences in number of solved problems for some categories and slightly less problems solved for Vampire*. The cause for this may stem from differences in the benchmarking (which is discussed in detail in [Bey16] and [BLW19]), a reduction in efficiency due to our implementation or other unknown factors.

Then, we enabled the method `VCNF` to see how this compares to the default method `FOOL2FOL`. Here we can see an increase in the number of solved problems in favor of `VCNF` which can be the result of directly defining the functions as opposed to `FOOL2FOL` where most functions are indirectly connected to their definitions through newly introduced function symbols as detailed in Chapter 5.

Looking at columns 3 and 4, we can compare Vampire* with and without function definition rewriting. Note that most of the recursive functions in the VIB set have intended orientations which coincide with most simplification orderings used by the superposition calculus. Some exceptions are the functions `mul` or `fltn`. So in theory, using paramodulation and disallowing inferences rewriting function definitions are to be blamed for most of the differences between these two variants of Vampire*. We can conclude that in this case losing refutational completeness this way really has a negative effect, as mostly just less problems are solved with function definition rewriting enabled.

Finally, the last column shows the results of function definition rewriting applied in case of Vampire**. Looking back at Figure 6.2, we can clearly see an increase in the number of solved problems overall and in the VIB set when function definition rewriting is enabled. However, UFDTLIA benchmarks show a more significant loss of solved problems.

Problems		Count	Vampire	Vampire*			Vampire**
Options					✓	✓	✓
						✓	✓
				−newcnf on			
				−fnrw on			
			Solved problems				
list	app	313	72	70	79	44	313
	pref	696	94	95	241	240	668
	rev	4	2	2	2	2	2
nat	add assoc	312	81	71	98	75	312
	add mix	830	103	106	161	176	510
	add comm	2	2	2	2	2	2
	add s 0 mix	486	81	83	92	96	252
	mul	6	0	0	0	0	0
	even	2	0	0	0	0	0
	equal	4	2	2	1	1	3
	leq	696	117	118	166	174	696
	dup	1	0	0	0	0	0
btree	fltn	3	0	0	0	0	0
nat +list	add +app	1	1	1	1	1	1
UFDTLIA		327	144	142	153	128	123
All		3683	699	692	996	939	2882

Problems		Count	Uniquely solved problems				
list	app	313	0	0	0	0	219
	pref	696	0	0	2	0	432
	rev	4	0	0	0	0	0
nat	add assoc	312	0	0	0	0	198
	add mix	830	0	0	6	4	323
	add comm	2	0	0	0	0	0
	add s 0 mix	486	0	0	0	0	150
	mul	6	0	0	0	0	0
	even	2	0	0	0	0	0
	equal	4	0	0	0	0	1
	leq	696	0	0	0	0	490
	dup	1	0	0	0	0	0
btree	fltn	3	0	0	0	0	0
nat +list	add +app	1	0	0	0	0	0
UFDTLIA		327	0	1	3	1	4
All		3683	0	1	11	5	1817

Figure 6.3: Preprocessing methods and function definition rewriting compared

We attribute this once again to the lack of variations in the VIB set which shows an improvement in almost all categories even though we use an incomplete approach.

6.6 Combining the techniques

First, we would like to emphasize that most of the techniques showed a tendency to uniquely solve problems in the experiments. Due to the relatively large number of introduced options we refrain from doing any experiments with more options enabled. Instead, we show how sets of options can be combined to solve more problems. For this, we use the *portfolio mode* of Vampire with a *custom schedule*. This mode essentially slices up the time limit given and in each a newly spawned Vampire instance is run with the next set of options given by the schedule. We created a schedule which runs Vampire** with all combinations of the most prominent options, namely `-indoct`, `-indstr`, `-indmc`, `-inds`, `-indtosh` and `-fnrw` on top of the default options `-sik four` and `-newcnf on`, each with a time limit of 6 seconds. This schedule gives us 64 sets of options that already take up more than 300 seconds so it is somewhat useless to include any other options. We denote this version of Vampire with Vampire†.

We ran experiments with other solvers as well, in particular with CVC4 and ZipperPosition, both of which gave promising results in the paper [HHK+20]. We used ZipperPosition version 2.1 with default flags and CVC4 version 1.8 with options `--quant-ind` and `--conjecture-gen` apart from the ones for setting the time limit and input/output formats. ZipperPosition does not support the SMT-LIB input format, so we could only use it to for experimenting with the VIB set which provides `zf` problem definitions with function definition blocks which is the most well-suited for this prover. We set the same limits for memory and time in these solvers, although a memory limit cannot be directly set in CVC4 (it was only enforced by BenchExec). Finally, we included an original Vampire and a Vampire** run with only default options for baseline. The resulting two tables can be found in Figure 6.4.

The tables show that in the VIB benchmark set, Vampire† and ZipperPosition gave the best results, with both solving a large number of problems uniquely. Although the overall number of solved problems is not comparable between the two solvers as ZipperPosition has no results in the UFDTLIA set, the difference within the VIB set is 365 in favor of Vampire† which is more than 10%.

In the UFDTLIA set, CVC4 is the clear winner with Vampire† in the second place only marginally increasing the best score within Vampire runs in this set (which was induction hypothesis strengthening in Figure 6.2).

Problems		Count	Solved problems				
			Vampire	Vampire**	Vampire†	CVC4	ZipperPosition
list	app	313	72	238	313	1	312
	pref	696	94	687	696	1	583
	rev	4	2	2	3	0	2
nat	add assoc	312	81	261	312	1	312
	add mix	830	103	263	591	70	552
	add comm	2	2	2	2	1	2
	add s 0 mix	486	81	124	281	87	186
	mul	6	0	0	2	0	0
	even	2	0	0	1	0	0
	equal	4	2	3	3	2	4
	leq	696	117	696	696	1	583
	dup	1	0	0	0	0	0
btree	fltn	3	0	0	2	1	2
nat +list	add +app	1	1	1	1	0	0
UFDTLIA		327	144	149	176	214	-
All		3683	699	2426	3079	379	2538

Problems		Count	Uniquely solved problems				
			Vampire	Vampire**	Vampire†	CVC4	ZipperPosition
list	app	313	0	0	1	0	0
	pref	696	0	0	0	0	0
	rev	4	0	0	1	0	0
nat	add assoc	312	0	0	0	0	0
	add mix	830	2	0	116	0	93
	add comm	2	0	0	0	0	0
	add s 0 mix	486	0	0	95	0	3
	mul	6	0	0	2	0	0
	even	2	0	0	1	0	0
	equal	4	0	0	0	0	0
	leq	696	0	0	0	0	0
	dup	1	0	0	0	0	0
btree	fltn	3	0	0	0	0	0
nat +list	add +app	1	0	0	0	0	0
UFDTLIA		327	2	0	9	50	-
All		3683	4	0	225	50	96

Figure 6.4: Vampire with and without portfolio mode and other solvers compared

CHAPTER 7

Conclusion

In this thesis, we have collected some of the most prominent techniques for *explicit induction on inductive datatypes using recursive function definitions*. These include recursion analysis techniques for *checking well-foundedness* (i.e. termination) of recursive function definitions with a lexicographic extension of subterm ordering, *identifying active function arguments* that participate in the recursion, *identifying recursive calls and branches* of each such function, then using this information to *create induction formulas, strengthen induction hypotheses, combine or discard induction formulas* when multiple choices are available. Using heuristics for *selecting induction term occurrences and complex induction terms*, we are able to prove a large set of problems.

Then, we have looked at specifically *saturation-based theorem proving* and how to adapt induction in this context: we added a new inference rule called *multi-clause induction* to be able to induct on more than one literal at a time and devised inference rules for *function definition rewriting* that align as much as possible with the properties of the superposition calculus to partially preserve completeness while allowing the prover to expand function headers in induction formulas into their definitions.

We implemented these techniques in the saturation-based first-order theorem prover Vampire. We also extended the prover's input format in order to parse recursive function definition blocks and implemented some limited heuristics to spot function definitions when given in an axiomatic way. Then, we performed experiments on a large set of inductive benchmarks including ones from SMT-LIB. We compared in these experiments the standard version of Vampire with various configurations of the new Vampire and also other first-order theorem provers and SMT solvers which showed how the techniques themselves are good starting points and – when properly combined through the portfolio mode of Vampire – give very good results compared to state-of-the-art theorem provers.

7.1 Future work

Most of the techniques listed in Chapter 3 are more from a heuristic approach than rigorously defined algorithms that work well on some problems but cannot tackle more complicated ones. By more complicated, we do not necessarily mean the size of the problem, rather the structural interconnectedness of the recursive function terms present in it. For example, the two induction term occurrence heuristics cover a large number of problems but fail in general. By more careful inspection of each function definition, one could determine exactly which occurrences are needed for the induction. The same goes even more when inducting on complex terms, where sophisticated methods can give better results because the work invested in finding only the most appropriate generalizations shows a return when the search space grows in a more controlled way.

It is also not clear how the induction schemes suggested by a term relate precisely to the ones suggested by its subterms, especially in the case when multiple induction schemes stem from one subterm. Here, techniques such as rippling can be helpful in deciding how to use those schemes.

Another open problem is finding the relation between induction hypotheses and their corresponding conclusions: even though the case distinction based on the function definition branching and discarding/combining induction schemes is correct w.r.t. the present function definitions, there is no guarantee that any of the induction hypotheses will actually be useful to prove the conclusions. A good example is the function `mul`, where the recursive case $\texttt{mul}(\texttt{s}(x), y) = \texttt{add}(\texttt{mul}(x, y), y)$ creates an additional `add` term upon expansion, therefore the induction hypotheses are often no use and only an additional induction step leads to a solution. Detecting such problems and adjusting (or discarding) the induction hypotheses accordingly in advance could reduce the search space greatly.

We have emphasized throughout the thesis that we handle a specific subset of problems where the functions do not contain side conditions for any branch or these can be folded into the function header and the well-foundedness of these functions admit some lexicographic order where each atomic ordering is the subterm order. As such, these require less work to show their well-definedness and well-foundedness. Upon extending the set of solvable problems to other problems, we will have to consider other techniques. For instance the quasi-commutation test devised by Walther is to show well-foundedness of arbitrary functions [Wal93, BD86].

Other issues arise when we consider the saturation-based approach. One is that the specific structure of the induction formulas converts to clausal forms where each case can be present in many of the result clauses, thus they are proven multiple times using a naive approach. AVATAR [Vor14] helps a lot in this matter but its splitting decisions do not necessarily coincide with the case distinction of an induction formula. Perhaps a more induction oriented splitting could reduce the number of proof steps. Another issue that is inherent in saturation-based theorem provers is not being able to rewrite the induction steps with the induction hypotheses as needed because the ordering prohibits it – as noted in Chapter 4 when introducing multi-clause induction. Unfortunately, using

the hypotheses as rewrite rules is not as straightforward as it was in the case of function definitions, since a priori we do not know which orientation will be required, not to mention that some hypotheses must be used multiple times during a single induction. Here, careful inspection of the problem at hand can also lead to more successful inductions.

The implementations we created in Vampire are far from efficient and it would be beneficial to take a fresh look at them in the future. Fortunately, induction is used seldom compared to other inference rules and is not considered a bottleneck, however, with an improved implementation maybe harder problems could be tackled. One would be to somehow precompute the induction schemes, e.g. by creating a special index that stores the suggested induction schemes with each subterm it contains – this would be useful mostly with multi-clause induction.

We conclude with a remark on the portfolio mode of Vampire. This mode allowed us to improve the results by trying multiple combinations of options in shorter time slices instead of just using one fixed set of options throughout a whole run. Although the improvement is far from substantial even with this naive enumeration of all option combinations, we emphasize that it can probably be improved when properly trained with special software to find the most suitable combinations for certain input problem sets.

Bibliography

[Ack28] W. Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928.

[Aub77] Raymond Aubin. Strategies for mechanizing structural induction. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'77, page 363–369, San Francisco, CA, USA, 1977. Morgan Kaufmann Publishers Inc.

[BBHI05] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2005.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BD86] Leo Bachmair and Nachum Dershowitz. Commutation, transformation, and termination. In Jörg H. Siekmann, editor, *8th International Conference on Automated Deduction*, pages 5–20, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[Bey16] Dirk Beyer. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 887–904, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

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

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 4(3):217–247, 06 1994.

[BHHW86] Susanne Biundo, B. Hummel, Dieter Hutter, and Christoph Walther. The Karlsruhe Induction Theorem Proving System. pages 672–674, 01 1986.

[BIS92] S. Baker, A. Ireland, and A. Smaill. On the use of the constructive omega-rule within automated deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning*, pages 214–225, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[BLW19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21, 02 2019.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press New York, 1979.

[BM88] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*. Academic Press Professional, Inc., USA, 1988.

[BM10] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BS10] James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216, 10 2010.

[Bun01] Alan Bundy. The Automation of Proof by Mathematical Induction. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 845–911. Elsevier and MIT Press, 2001.

[BvHHS90] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The OYSTER-CLAM system. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[CJRS13] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating Inductive Proofs Using Theory Exploration. pages 392–406, 06 2013.

[Cru17] Simon Cruanes. Superposition with Structural Induction. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasilia, Brazil, September 27-29, 2017, Proceedings*, volume 10483 of *Lecture Notes in Computer Science*, pages 172–188. Springer, 2017.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279 – 301, 1982.

[DF03] Lucas Dixon and Jacques Fleuriot. IsaPlanner: A Prototype Proof Planner in Isabelle. volume 2741, pages 279–283, 07 2003.

[DKM05] J. Dick, J. Kalmus, and U. Martin. Automating the Knuth-Bendix ordering. *Acta Informatica*, 28:95–119, 2005.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[EP20] M. Echenim and Nicolas Peltier. Combining Induction and Saturation-Based Theorem Proving. *Journal of Automated Reasoning*, 64(2):253–294, February 2020.

[GÖ31] K. Gödel. Über Formal Unentscheidbare Sätze der Principia Mathematica Und Verwandter Systeme I. *Monatshefte für Mathematik*, 38(1):173–198, 1931.

[H+78] Gérard Huet et al. On the uniform halting problem for term rewriting systems. 1978.

[HHK+20] Petra Hozzová, Márton Hajdú, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Induction with Generalization in Superposition Reasoning. In Christoph Benzmüller and Bruce Miller, editors, *Intelligent Computer Mathematics*, pages 123–137, Cham, 2020. Springer International Publishing.

[HKV11] Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Invariant Generation in Vampire. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 60–64, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple Word Problems in Universal Algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263 – 297. Pergamon, 1970.

[KKSV16] Evgenii Kotelnikov, Laura Kovács, Martin Suda, and Andrei Voronkov. A Clausal Normal Form Translation for FOOL. In Christoph Benzmüller, Geoff Sutcliffe, and Raúl Rojas, editors, *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, volume 41 of *EPiC Series in Computing*, pages 53–71. EasyChair, 2016.

[KKV15] Evgenii Kotelnikov, Laura Kovács, and Andrei Voronkov. A First Class Boolean Sort in First-Order Theorem Proving and TPTP. *CoRR*, abs/1505.01682, 2015.

[KL80] Samuel N. Kamin and J. Lévy. Two generalizations of the recursive path ordering. 1980.

[KP13] Abdelkader Kersani and Nicolas Peltier. Combining Superposition and Induction: A Practical Realization. 09 2013.

[KRV16] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to Terms with Quantified Reasoning. *CoRR*, abs/1611.02908, 2016.

[KV03] Konstantin Korovin and Andrei Voronkov. Orienting rewrite rules with the Knuth–Bendix order. *Information and Computation*, 183(2):165 – 186, 2003. 12th International Conference on Rewriting Techniques and Applications (RTA 2001).

[KV13] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[LVJ15] Irene Lobo Valbuena and Moa Johansson. Conditional Lemma Discovery and Recursion Induction in Hipster. 09 2015.

[Mid] Aart Middeldorp. Term Rewriting Lecture Notes.

[MW13] J. Strother Moore and Claus-Peter Wirth. Automation of Mathematical Induction as part of the History of Logic. 09 2013.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. 01 2002.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, pages 337–351, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[RK15] Andrew Reynolds and Viktor Kuncak. Induction for SMT Solvers. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 80–98, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[RSV16] Giles Reger, Martin Suda, and Andrei Voronkov. New Techniques in Clausal Form Generation. In Christoph Benzmüller, Geoff Sutcliffe, and Raúl Rojas, editors, *GCAI 2016. 2nd Global Conference on Artificial Intelligence, September 19 - October 2, 2016, Berlin, Germany*, volume 41 of *EPiC Series in Computing*, pages 11–23. EasyChair, 2016.

[RV01] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[RV19] Giles Reger and Andrei Voronkov. Induction in saturation-based proof search. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 477–494. Springer, 2019.

[SDE12] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An automated prover for properties of recursive data structures. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 407–421, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[SJC15] Nick Smallbone, Moa Johansson, and Koen Claessen. Tons of Inductive Problems (TIP). `tip-org.github.io`, 2015.

[Sut17] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[Tea09] The Coq Development Team. The Coq Proof Assistant Reference Manual, 2009.

[Tur37] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 01 1937.

[Vor14] Andrei Voronkov. AVATAR: The Architecture for First-Order Theorem Provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 696–710, Cham, 2014. Springer International Publishing.

[Wal92] Christoph Walther. Computing induction axioms. pages 381–392, 07 1992.

[Wal93] Christoph Walther. Combining induction axioms by machine. pages 95–101, 01 1993.

[Wal94] Christoph Walther. *Mathematical Induction*, pages 127–228. 04 1994.