Superposition: Types and Induction

A dissertation submitted towards the degree
Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

by
Daniel Wand

Saarbrücken
May, 2017

Tag des Kolloquiums:	4ter August 2017
Dekan:	Prof. Dr. Frank-Olaf Schreyer
Mitglieder des Prüfungsauschuss	
Vorsitzender:	Prof. Dr. Holger Hermanns
Berichterstatter:	Prof. Dr. Christoph Weidenbach
	Dr. Jasmin C. Blanchette
	Prof. Geoff Sutcliff, Ph.D.
Akademischer Mitarbeiter:	Dr. Swen Jacobs

Abstract

Proof assistants are becoming widespread for formalization of theories both in computer science and mathematics. They provide rich logics with powerful type systems and machine-checked proofs which increase the confidence in the correctness in complicated and detailed proofs. However, they incur a significant overhead compared to pen-and-paper proofs.

This thesis describes work on bridging the gap between high-order proof assistants and first-order automated theorem provers by extending the capabilities of the automated theorem provers to provide features usually found in proof assistants.

My first contribution is the development and implementation of a first-order superposition calculus with a polymorphic type system that supports type classes and the accompanying refutational completeness proof for that calculus. The inclusion of the type system into the superposition calculus and solvers completely removes the type encoding overhead when encoding problems from many proof assistants.

My second contribution is the development of SupInd, an extension of the typed superposition calculus that supports data types and structural induction over those data types. It includes heuristics that guide the induction and conjecture strengthening techniques, which can be applied independently of the underlying calculus.

I have implemented the contributions in a tool called Pirate. The evaluations of both contributions show promising results.

Zusammenfassung

Beweisassistenten werden zunehmend in der Formalisierung von Theorien, sowohl in der Informatik als auch in der Mathematik, genutzt. Ihre vielseitigen Logiken mit ausdrucksstarken Typsystemen ermöglichen maschinenkontrollierte Beweise. Dies erhöht die Vertrauenswürdigkeit von komplizierten und detaillierten Beweisen. Im Gegensatz zu Papierbeweisen ist ihr Gebrauch jedoch aufwendiger.

Diese Dissertation beschreibt Fortschritte darin, den Abstand zwischen Beweisassistenten höherer Stufe und automatischen Beweissystemen erster Stufe zu schließen, indem automatische Beweissysteme so erweitert werden, dass sie die Möglichkeiten die Beweisassistenten bieten auch bereit stellen.

Der erste Beitrag ist die Erweiterung des Superpositionskalküls erster Stufe um ein polymorphes Typsystem, das Typklassen unterstützt. Die Erweiterung beinhaltet einen Beweis der Widerlegungsvollständigkeit. Das Typsystem als Teil des Superpositionskalkül ermöglicht die Übertragung von Problemen aus Beweisassistenten ohne den sonst üblichen Mehraufwand durch Typenenkodierung.

Der zweite Beitrag ist die Entwicklung von SupInd, einer Erweiterung von Superposition, die Datentypen und strukturelle Induktion über diese Datentypen ermöglicht. SupInd beinhaltet Heuristiken, die die Induktion lenken und Annahmenverstärkungstechniken, die auch unabhängig des Kalküls benutzt werden können.

Die Beiträge wurden im Tool Pirate umgesetzt, die Evaluationen zeigen vielversprechende Ergebnisse.

Acknowledgements

I would like to thank my thesis advisor Christoph Weidenbach for his constant support, guidance and for introducing me to the world of logic and automated reasoning. I am thankful for the opportunity to work in his group, where I had the freedom to pursue both the theoretical and practical aspects of automated reasoning. I am grateful to my thesis co-advisor Jasmin Christian Blanchette for his numerous discussions and for sharing his expertise on Isabelle and Sledgehammer. I would also like to thank Geoff Sutcliffe, who kindly agreed to review this thesis.

Working on the thesis and with my colleagues in both the Automation of Logic group and the Theorem Proving Group in Munich was an enriching experience. Special thanks go to Tobias Nipkow, for hosting me several times in Munich and allowing me to work for the TU Munich for a year, Uwe Waldmann, for the many times he explained the details of superposition to me, Arnaud Fietzke and Martin Suda, for helping me get started with induction and superposition, Patrick Wischnewski, for his invaluable help navigating and understanding the SPASS source code, Willem Hagemann, for being a sounding board for ideas and philosophical-logical discussions, Martin Bromberger and Mathias Fleury, for proof reading drafts of this thesis, Gabor Alagi, Noran Azmy, Marek Kosta, Evgeny Kruglov, Thomas Sturm, Andrei Popescu, Ching Hoo Tang, Andreas Teucke, Marco Vogt and Tianxiang Lu, for many interesting discussions and interactions. I also want to thank my wife Laura for her love and support.

I would also like to thank the Max Planck Society and the Max Planck Institute for Informatics in particular for the great research environment and resources. Without the computing cluster none of my evaluations would have been possible in their current form.

This work was partly supported by International Max Planck Research School, the Saarbrücken Graduate School of Computer Science and the German Research Foundation (DFG), in particular by the DFG Priority Programme 1496 – ‘Reliably Secure Software Systems (RS3)’ and the DFG Project ‘Hardening the Hammer’.

Contents

1. Introduction

This thesis describes work on bridging the gap between high-order proof assistants and first-order automated theorem provers by extending the capabilities of automated theorem provers to provide features generally found in proof assistants.

1.1. Motivation

Proof assistants are becoming widespread for formalization of theories both in computer science and mathematics. They provide rich logics with powerful type systems and machine-checked proofs which increase the confidence in the correctness in complicated and detailed proofs. However, despite their rising popularity, proof assistants can have a significant overhead compared to theorem proving with pen-and-paper and require significant interaction of the user of such an proof assistant.

Automated theorem provers on the other hand provide push-button automation to theorem proving that requires no user interaction. Their main disadvantage is that the most successful of them can only prove theorems in (untyped) first-order logic without many of the conveniences and features that proof assistants provide. As part of the preparatory phase of the Graduate School of Computer Science, I extended SPASS to support a monomorphic type system [9]. SMT-solvers and superposition provers like Vampire also support monomorphic type systems [21, 23, 39, 50].

Proof assistants can harness the power of automated theorem provers by encoding their rich structure into the first-order logic that automated theorem provers can process [2, 11]. This process has two main phases. First, higher-order features are encoded into first-order constructors. After the removal of higher-order features, the proof assistant's type system has to be encoded into a type system that the automated theorem prover supports.

This has several drawbacks. For one, it is generally acknowledged that encodings add extra structure into the problem which then complicates the tasks for the automated theorem provers [20]. This can drastically reduce their success rate. Even worse, not all encodings lead to proofs that can be reconstructed in a proof assistant [11]. For example, the simple definition of reverse over the data type of list over natural numbers can be defined as the following, where @ is the list append written in infix notation and [...] denotes the singleton list:

$$\textbf{data type}\ \mathcal{N}\text{-list} : \mathit{nil} \mid \mathit{cons}(\mathcal{N},\mathcal{N}\text{-list})$$

$$\mathit{rev}(\mathit{nil}) = \mathit{nil}$$

$$\mathit{rev}(\mathit{cons}(X_{\mathcal{N}}, \mathit{XS}_{\mathcal{N}\text{-list}})) = \mathit{rev}(\mathit{XS}_{\mathcal{N}\text{-list}})\,@\,[X_{\mathcal{N}}]$$

When encoding this to untyped first-order logic (via the so called type guards encoding) two predicates ($p_{\mathcal{N}}$ and $p_{\mathcal{N}\text{-list}}$), must be introduced to encode if a term is of the natural numbers or a list. Additionally, axioms must be added to specify which terms are of the type the predicate corresponds to. The two simple equations encoded into untyped first-order logic then become the following formulas:

$$p_{\mathcal{N}\text{-list}}(nil)$$

$$p_{\mathcal{N}}(X) \wedge p_{\mathcal{N}\text{-list}}(XS) \Rightarrow p_{\mathcal{N}\text{-list}}(cons(X, XS))$$

$$rev(nil) = nil$$

$$p_{\mathcal{N}}(X) \wedge p_{\mathcal{N}\text{-list}}(XS) \Rightarrow rev(cons(X, XS)) = rev(XS)@[X]$$

$$p_{\mathcal{N}}(X) \wedge p_{\mathcal{N}\text{-list}}(XS) \Rightarrow p_{\mathcal{N}\text{-list}}(rev(X, XS))$$

This encoding also loses information. In a proof assistant, the two equations that define the list reverse (*rev*) are implicitly oriented from left to right, i.e. in the order of *rev*'s computation. This information about orientation is lost by the translation into untyped fist-order logic. Even worse, the recursive equation turns into a clause with three literals, making simplification via simple equational rewriting impossible. Furthermore, the information that $\mathcal{N}$-list is a data type is also lost. Even by adding further formulas, that information can only be partially encoded, because induction (an essential property of data types) cannot be finitely first-order axiomatized.

The challenge posed by the significant overhead that encodings involve is further strengthened by the fact that automated theorem provers are designed to find theorems, but have trouble showing that sets of axioms are satisfiable. Whereas proof assistants generally have a vast set of axioms where only a small (a priori unknown) subset is relevant to proving a theorem. Thus, any proof obligation passed to an automated theorem prover will contain hundreds of irrelevant axioms and a handful of necessary ones. Ideally, we want to pass as many axioms as possible to the automated theorem provers, because it is hard to determine which axioms are irrelevant. The fewer encodings an automated theorem prover has to handle, the more likely it is that it does explore the relevant axioms, thereby finding a proof.

As mentioned, a further difference between automated theorem provers and proof assistance is the support of data types and induction over them. For example, the property that the reverse of the reverse of a list is the equal to the original list

$$rev(rev(XS)) = XS$$

is not a first-order consequence of the defining equations of *rev* but it is an inductive consequence. Since induction cannot be finitely axiomatized, support of induction inside the automated theorem provers is useful for any integration into a proof assistant.

1.2. Contributions

This thesis describes work on two main contributions toward bridging the gap between proof assistants and automated theorem provers.

The first contribution of this thesis is the development and implementation of a first-order superposition calculus with a polymorphic type system that supports type classes and the accompanying refutational completeness proof for that calculus. The inclusion of the type system into the superposition calculus removes the type encoding overhead when encoding problems from proof assistants such as the high-order logic proof assistant family (HOL) to this version of superposition (the encoding overhead for higher-order features remain). The HOL family is comprised of, among others, Isabelle/HOL, HOL4, HOL Light and ProofProver–HOL [25, 28, 49]. Furthermore, I proved that superposition extended with the polymorphic type system that supports type classes is still refutationally complete. The evaluation of my implementation — Pirate — shows that the polymorphic calculus is superior to the usage of type encodings. The evaluation also shows that the polymorphic calculus it is competitive with it the simpler monomorphic type system, even though it is more expressive than the monomorphic encoding.

My second contribution is the development of the SupInd calculus, an extension of typed superposition that allows for recursive data types and structural induction over those data types. SupInd features a novel heuristic that guides the application of induction to promising positions while employing a technique that mitigates the result of irrelevant induction steps. I have also developed two new conjecture strengthening heuristics (and adapted an existing one) that can be applied independently of the underlying SupInd calculus. I also present an optimized variant of SupInd that avoids (some) unnecessary inferences and is capable of detecting and removing invalid conjectures and propositions. I have implemented SupInd in a specialized version of Pirate with promising evaluation results on two standard benchmark sets.

1.3. Structure of This Thesis

The structure of this thesis is as follows:

- Chapter 2 introduces the necessary background on first-order logic as well as the superposition calculus, which underlies many successful automated theorem provers.
- Chapter 3 describes the polymorphic first-order logic, including syntax with unification, typing rules, semantics and the necessary transformations using clausification and Skolemization. It also contains the proof that superposition is refutationally complete for that polymorphic logic.
- Chapter 4 describes SupInd, an extension of superposition to handle structural induction and conjecture strengthening techniques. It also includes the discussion on the developed heuristics and strengthening techniques.
- Chapter 5 summarizes my results and outlines future research opportunities.

Even though induction extends on the type system chapter, chapters 3 and 4 can be read in any order. The evaluations and related work is presented at the end of each chapter.

2. Preliminaries

The rest of this thesis requires knowledge of first-order logic and superposition [5], which will be introduced in this chapter. The first-order logic introduced in this chapter is the standard, untyped version of first-order logic. It is implemented by many automated theorem provers. The text of this introductory section is heavily based on Weidenbach's script of the automated reasoning lecture given at Saarland University [38] with some adaptions from Bachmair and Ganzinger [5].

First, I will present the untyped first-order logic language with build-in equations (Sect. 2.1) and then the superposition calculus, including the necessary machinery and the refutational completeness proof (Sect. 2.2). The refutational completeness proof for superposition will then be extended to the typed case in chapter 3 and further with (structural) induction in chapter 4.

2.1. Untyped First-Order Logic

In this section, I formally describe the syntax and semantics of untyped, equational first-order logic. I will not provide the rules for predicates in our logic as both their rules are similar to those of functions and it is generally known that predicates can be expressed as functions combined with a special truth value.

2.1.1. Syntax

The syntax determines the allowed expressions and how they are built. It consists of the signature that declares the used symbols and their properties and the rules how to form terms, clauses and formulas and the syntactic operations of substitution and unification.

Definition 2.1.1.1 (**Signature**). A *signature* fixes a set of non-logical symbols used to construct the terms and formulas. A signature Σ for untyped, equational first-order is the set $S_{\mathcal{F}}$, where

- $S_{\mathcal{F}}$ is a set of function symbols f with arity ≥ 0, written as $\text{arity}(f) = m$;
- $S_{\mathcal{P}}$ is a set of predicate symbols p with arity ≥ 0, written as $\text{arity}(p) = m$;

From the symbols in the definitions of the signature terms can be build.

Definition 2.1.1.2 (**Terms**). Let Σ be a signature and $\mathcal{X}$ be a given at most countably infinite set of term variables. Then all *untyped terms* $T_\Sigma(\mathcal{X})$ are recursively defined as follows:

1. Every term variable in $\mathcal{X}$ is a term.
2. For all function symbols $f \in S_{\mathcal{F}}$ with $arity(f) = m$ if $t_1,\ldots,t_m$ are terms then $f(t_1,\ldots,t_m)$ is also a term.

A term is *ground* if it contains no variables. A function (symbol) is a *constant* if it has no arguments, for terms only consisting of a constant I also use f instead of $f()$. I will use s, t for terms, f for function symbols and u, v for term variables.

A *position* p in a term t is a list of natural numbers $[p_1, \dots, p_n]$ and is defined as $\text{pos}([p_1, \dots, p_n], f(t_1, \dots, t_m)) = \text{pos}([p_2, \dots, p_n], t_{p_1})$ and $\text{pos}([], t) = t$. I write $t[s]_p$ to denote that the term s is placed at position p in another term t.

Definition 2.1.1.3 (Formulas). Let Σ be a signature. Then all *untyped formulas* $F_\Sigma(\mathcal{X})$ are recursively defined as follows:

1. $\bot$	
2. $\top$	
3. $p(t_1, \dots, t_m)$	if $p \in S_\mathcal{P}$, $t_1, \dots, t_m \in T_\Sigma(\mathcal{X})$ and $\text{arity}(p) = m$
4. $s \approx t$	if $s, t \in T_\Sigma(\mathcal{X})$
5. $\neg\phi$	if $\phi \in F_\Sigma(\mathcal{X})$
6. $\phi \wedge \psi$	if $\phi, \psi \in F_\Sigma(\mathcal{X})$
7. $\phi \vee \psi$	if $\phi, \psi \in F_\Sigma(\mathcal{X})$
8. $\phi \rightarrow \psi$	if $\phi, \psi \in F_\Sigma(\mathcal{X})$
9. $\phi \leftrightarrow \psi$	if $\phi, \psi \in F_\Sigma(\mathcal{X})$
10. $\forall u.\, \phi$	if $\phi \in F_\Sigma(\mathcal{X})$
11. $\exists u.\, \phi$	if $\phi \in F_\Sigma(\mathcal{X})$

Let Q be either $\forall$ or $\exists$. Then for a (sub)formula $Qu.\, \phi$ the variable u is *bound* within ϕ. A formula where all variables are bound, i.e. have a corresponding quantifier, is called *closed*. The formulas constructed by 1–4 are *atomic formulas*. Atomic formulas and their negations ($\neg$) are *literals*.

Definition 2.1.1.4 (Clauses). A *clause* is a disjunction of one or more literals without any quantifiers. I consider all variables of a clause to be implicitly universally quantified.

The clause consisting only of $\bot$ is called the *empty clause*. Formulas and clauses are *ground* if they contain no variables. I will use ϕ, ψ for formulas, C, D for clauses and N for clause sets.

Substitutions Here, I introduce substitutions for variables, terms and clauses.

Definition 2.1.1.5 (Variable Substitutions). A *substitution* is a mapping from variables ($\mathcal{X}$) to terms ($T_\Sigma(\mathcal{X})$).

I will use σ, θ for substitutions and write them as $\sigma = \{u_1 \mapsto t_1, \dots\}$, where all u_i are pairwise distinct, and define them to be:

$$\sigma(v) = \begin{cases} t_i & \text{if } v = u_i \\ v & \text{otherwise} \end{cases}$$

I also write $v\sigma$ for $\sigma(v)$. A substitution σ is updated to return t for u by

$$\sigma[u \mapsto t](v) = \begin{cases} t & \text{if } v = u \\ \sigma(v) & \text{otherwise} \end{cases}$$

Definition 2.1.1.6 (Term Substitutions). Substitutions are extended to non-variable terms by

$$f(t_1,\, \ldots,\, t_m)\sigma = f(t_1\sigma,\, \ldots,\, t_m\sigma)$$

Definition 2.1.1.7 (Clause Substitutions). Substitutions are similarly extended to clauses by

1. $\bot\sigma = \bot$
2. $\top\sigma = \top$
3. $p(t_1,\, \ldots,\, t_m)\sigma = p(t_1\sigma,\, \ldots,\, t_m\sigma)$
4. $(s \approx t)\sigma = s\sigma \approx t\sigma$
5. $(\neg(s \approx t))\sigma = \neg(s\sigma \approx t\sigma\ )$
6. $(\phi \lor \psi)\sigma = (\phi\sigma \lor \psi\sigma)$

I do not need substitutions for formulas so I only define them for clauses.

Unification *Unification* describes the obligation to find a substitution σ such that for two terms s and t it holds that $s\sigma = t\sigma$. A substitution σ_1 is a more general unifier than the substitution σ_2 if there exists a substitution σ_3 such that for all terms t: $t\sigma_2 = t\sigma_1\sigma_3$. A substitution is the *most general unifier* (*mgu*) if for all σ_2 there exists a substitution σ_3 such that for all terms t: $t\sigma_2 = t\sigma_1\sigma_3$. It is well known that if two (untyped first-order) terms are unifiable, then they have a unique most general unifier (up to renaming of the variables).

2.1.2. Semantics

The semantics gives meaning to the constructs that can be build with a given syntax. It consists of the structure and the interpretations. I will also define the notion of satisfiability here.

Definition 2.1.2.1 (Structure). A Σ-*structure* is a tuple $(U,\, I_F,\, I_P)$, where

- $U \neq \emptyset$ is the universe, a non-empty set
- I_F is the set of functions $f^{\mathcal{I}} : U^m \to U$ for each function symbol $f \in S_F$, with $\text{arity}(f) = m$
- I_P is the set of predicates $p^{\mathcal{I}} : U^m \to \{0,\, 1\}$ for each predicate symbol $p \in S_P$, with $\text{arity}(p) = m$

I will use e for elements of U.

Interpretations The meaning of a variable is given by a valuation. A *variable valuation* ($\mathcal{V}$) is a mapping $\mathcal{X} \to U$ and is always relative to a Σ-structure. I leave this relation implicit. A valuation $\mathcal{V}$ is updated to return e for u by

$$\mathcal{V}[u \mapsto e](v) = \begin{cases} e & \text{if } v = u \\ \mathcal{V}(v) & \text{else} \end{cases}$$

Definition 2.1.2.2 (Interpretations of Terms). Valuations are extended to *interpretations* of terms ($T_\Sigma(\mathcal{X}) \to U$) by the following

$$\begin{array}{lcl}
\mathcal{I}_\mathcal{V}(u) & = & V(u) \\
\mathcal{I}_\mathcal{V}(f(t_1, \ldots, t_m)) & = & f^\mathcal{I}(\mathcal{I}_\mathcal{V}(t_1), \ldots, \mathcal{I}_\mathcal{V}(t_m))
\end{array}$$

where $f^\mathcal{I} \in I_F$ is the function corresponding to the function symbol $f \in S_\mathcal{F}$ and m its arity (and equivalently for $p^\mathcal{I} \in I_P$).

An interpretation $\mathcal{I}_\mathcal{V}$ is *term-generated*, if for every $e \in U$ there exists a ground term $t \in T_\Sigma(\emptyset)$ such that $e = \mathcal{I}_\mathcal{V}(t)$.

Definition 2.1.2.3 (Interpretations of Formulas). Based on the interpretations of terms, the interpretation of formulas is given by the following

$$\begin{array}{lcl}
\mathcal{I}_\mathcal{V}(\bot) & = & 0 \\
\mathcal{I}_\mathcal{V}(\top) & = & 1 \\
\mathcal{I}_\mathcal{V}(p(t_1, \ldots, t_m)) & = & p^\mathcal{I}(\mathcal{I}_\mathcal{V}(t_1), \ldots, \mathcal{I}_\mathcal{V}(t_m)) \\
\mathcal{I}_\mathcal{V}(s \approx t) & = & 1 \Leftrightarrow \mathcal{I}_\mathcal{V}(s) = \mathcal{I}_\mathcal{V}(t) \\
\mathcal{I}_\mathcal{V}(\neg\phi) & = & 1 - \mathcal{I}_\mathcal{V}(\phi) \\
\mathcal{I}_\mathcal{V}(\phi \land \psi) & = & \min(\mathcal{I}_\mathcal{V}(\phi), \mathcal{I}_\mathcal{V}(\psi)) \\
\mathcal{I}_\mathcal{V}(\phi \lor \psi) & = & \max(\mathcal{I}_\mathcal{V}(\phi), \mathcal{I}_\mathcal{V}(\psi)) \\
\mathcal{I}_\mathcal{V}(\phi \to \psi) & = & \max(1 - \mathcal{I}_\mathcal{V}(\phi), \mathcal{I}_\mathcal{V}(\psi)) \\
\mathcal{I}_\mathcal{V}(\phi \leftrightarrow \psi) & = & 1 \Leftrightarrow \mathcal{I}_\mathcal{V}(\phi) = \mathcal{I}_\mathcal{V}(\psi) \\
\mathcal{I}_\mathcal{V}(\forall x.\phi) & = & \min\limits_{e \in U}(\mathcal{I}_{\mathcal{V}[x \mapsto e]}(\phi)) \\
\mathcal{I}_\mathcal{V}(\exists x.\phi) & = & \max\limits_{e \in U}(\mathcal{I}_{\mathcal{V}[x \mapsto e]}(\phi))
\end{array}$$

A formula ϕ is *valid* in the interpretation $\mathcal{I}_\mathcal{V}$ ($\mathcal{I}_\mathcal{V}$ is a *model* of ϕ) if $\mathcal{I}_\mathcal{V} \vDash \phi \Leftrightarrow \mathcal{I}_\mathcal{V}(\phi) = 1$. A formula ϕ *entails* a formula ψ ($\phi \vDash \psi$) if for all interpretations $\mathcal{I}_\mathcal{V}$ it holds that $\mathcal{I}_\mathcal{V} \vDash \phi$ implies $\mathcal{I}_\mathcal{V} \vDash \psi$. A formula ϕ is *satisfiable* if and only if there exists an interpretation $\mathcal{I}_\mathcal{V}$ such that $\mathcal{I}_\mathcal{V} \vDash \phi$. If there exists no such interpretation the formula is unsatisfiable. I call an interpretation a *model* of a clause (formula) set N if and only if all clauses (formulas) of that set are true under that interpretation.

2.1.3. Clausification and Skolemization

Superposition is defined on implicitly universally quantified clauses. Thus, in order to apply superposition to arbitrary formulas existential quantification must be removed and the formulas must be transformed to clauses.

A formula is in *prenex normal form* if all quantifiers are at the top of a quantifier-free formula, i.e. the formula has the form $Q_1 u_1. \ldots Q_n u_n. \phi$ where each Q_i is either $\exists$ or $\forall$ and ϕ is quantifier-free. Any formula can be translated ($\Longrightarrow_P$) to an equivalent formula in prenex normal form. *Skolemization* replaces the existential quantifiers by a *choice function* (often called *Skolem function*), which computes the existential quantifier. The choice of what the existential quantifier chooses depends only on the quantifiers above it or alternatively on the all free variables below it.

Those variables turn into the arguments of the Skolem function. I apply Skolemization first to the top most existential quantifier of a formula in prenex normal form. This existential quantifier only depends on the universal quantifiers above it. The Skolem transformation ($\Longrightarrow_S$) then is

$$\forall u_1 \ldots \forall u_m.\, \exists u.\phi \Longrightarrow_S \forall u_1 \ldots \forall u_m.\, \phi\{u \mapsto f(u_1,\, \ldots,\, u_m)\}$$

where $n \geq 0$ and f is a fresh (Skolem) function symbol of arity m that is not present in the signature ($S_{\mathcal{F}}$). The transformation also adds the function symbol f to the signature. The resulting formula of the Skolem transformation is not equivalent to the original formula, because the signature has changed. It is guaranteed that the original formula is satisfiable in the original signature if and only if the formula produced by the Skolem transformation is satisfiable in the extended signature. *Clause normal form* transformation ($\Rightarrow_{CNF}$) translates a prenex normal formula without existential quantifiers into an equivalent set of clauses. The set of clauses is then interpreted as an n-ary conjunction. I will not focus on clausification but note that any set of formulas can be transformed in an equisatisfiable set of clauses by transforming each formula via performing prenex normal form transformation followed by exhaustive application of Skolemization and then applying clause normal form transformation.

2.2. Superposition

Superposition [5] is one of the most successful calculi for untyped equational first-order logic. It is refutationally complete for first-order logic and a decision procedure for various first-order logic fragments. Superposition is usually untyped, but there are variants which support (monadic) sorts, e.g. the version underlying SPASS [67]. Because of it success in automatically solving first-order problems, superposition-based theorem provers are important backend tools to attempts to automate higher-order proof assistants, e.g. for tools such as Sledgehammer [10]. The one guiding theme of this research is to make superposition more suitable for such a integration.

2.2.1. Literal and Clause Orderings

The superposition calculus is parameterized by orderings on terms, literals and clauses [5]. They are used to decrease the necessary search space by restricting the inferences to the largest literals and terms.

Definition 2.2.1.1 (**Admissible Literal Ordering**). Following Bachmair and Ganzinger's definition [5], an ordering on literals $\succ$ is *admissible* if

1. it is total on ground literals,
2. its restriction to terms is a reduction ordering and
3. *closed under substitutions*: for all literals L and L' and all substitutions σ it holds that $L \succ L'$ implies $L\sigma \succ L'\sigma$
4. for all literals L and L' it holds that $L \succ L'$ if
    a) $\max(L) \succ \max(L')$ where max is the maximal term of a literal or

b) $\max(L) = \max(L')$ where L is negative and L' is positive

I call a literal L in a clause $C' \vee L$ *maximal* if there is a substitution σ such that $L\sigma \succeq L'$ for all literals L' in C'. I call a literal L in a clause $C' \vee L$ *strictly maximal* if there is a substitution σ such that $L\sigma \succ L'$ for all literals L' in C'.

Definition 2.2.1.2 (Admissible Clause Ordering). Again, following Bachmair and Ganzinger definition [5], an ordering on clauses $\succ$ is *admissible* if

1. it is well-founded,
2. irreflexive and
3. *closed under substitutions*: for all clauses C and C' and all substitutions σ it holds that $C \succ C'$ implies $C\sigma \succ C'\sigma$
4. for all clauses C and C' it holds that $C \succ C'$ if
   a) $\max(C) \succ \max(C')$ where max is the maximal literal of a clause.

2.2.2. Term Orderings

Term orderings are used to restrict the direction of superposition inferences to replacing terms only with other terms that are not larger. The term orderings that are used are usually lifted to an ordering on literals by assigning the literal $s \approx t$ the multiset $\{s,t\}$ and the literal $s \not\approx t$ (also written as $\neg s \approx t$) the multiset $\{s,s,t,t\}$. Then the multiset extension of the term ordering is used for the literal ordering. Similarly, clauses are ordered by considering them as multisets of literals and using the multiset extension of the literal ordering. Clause and literal ordering lifted in this way from reduction orderings are admissible [5].

Reduction Orderings

A reduction ordering (as defined by Baader and Nipkow [3, p. 102]) is a well-founded rewrite ordering. A total reduction ordering is also a simplification ordering, i.e. subterms are smaller than superterms. A (total) reduction ordering thus is a (total) well-founded strict ordering $\succ$ and is

1. (*total*: for all s_1, s_2,
   $s_1 \succ s_2 \vee s_2 \succ s_1 \vee s_1 = s_2$)
2. *irreflexive*: for all s_1,
   $\neg(s_1 \succ s_1)$
3. *transitive*: for all s_1, s_2, s_3,
   $s_1 \succ s_2 \wedge s_2 \succ s_3 \implies s_1 \succ s_3$
4. *compatible with Σ-operations*: for all $t_1, \ldots, t_m, s_1, s_2 \in T_\Sigma(\mathcal{X})$ and all $m \geq 0$ and all function symbols f of arity m,
   $s_1 \succ s_2$ implies $f(t_1, \ldots, s_1, \ldots, t_m) \succ f(t_1, \ldots, s_2, \ldots, t_m)$

5. *closed under substitutions*: for all $s_1,\ s_2 \in T_\Sigma(\mathcal{X})$ and all substitutions σ,
   $s_1 \succ s_2$ implies $s_1\sigma \succ s_2\sigma$

Simplification Orderings

A simplification ordering is a reduction ordering that has the subterm property, i.e. all subterms are smaller than their superterms.

Knuth-Bendix Ordering The Knuth-Bendix Ordering (KBO) [37] is a widely used simplification ordering, especially in superposition-based automated theorem provers. I present the KBO adapted from Baader and Nipkow [3, p. 124].

Let $|t|_u$ denotes how often the variable u occurs in the term t. Let $>_\mathrm{f}$ be a strict ordering on the function symbols and let w be a weight function with the following properties:

1. There exists $w_0 \in \mathbb{R}^+ \backslash \{0\}$ such that for all variables u, $w(u) = w_0$ for all constant symbols f, $w(f) \geq w_0$
2. If f is a unary function symbol with weight $w(f) = 0$, then f is the greatest element with respect to $>_\mathrm{f}$

The weight function w is extended to terms by $w(f(t_1,\ \ldots,\ t_n)) = w(f) + w(t_1) + \cdots + w(t_n)$.
I define s $\succ_{KBO}$ t to hold if and only if:

1. For all term variables u: $|s|_u \geq |t|_u$ and $w(s) > w(t)$, or
2. For all term variables u: $|s|_u \geq |t|_u$ and $w(s) = w(t)$, and one of the following:
   a) There exists a unary function symbol f, a variable u and a positive integer n such that $s = f^n(u)$ and $t = u$
   b) There exist functions symbols f, g such that $f >_\mathrm{f} g$ and $s = f(s_1,\ \ldots,\ s_{n_f})$ and $t = g(t_1,\ \ldots,\ t_{n_g})$
   c) There exists a function symbol f and an index i, $1 \leq i \leq n$, such that $s = f(s_1,\ \ldots,\ s_n)$, $t = f(t_1,\ \ldots,\ t_n)$ and $s_1 = t_1,\ \ldots,\ s_{i-1} = t_{i-1}$ and $s_i \succ_{KBO} t_i$

For untyped first-order superposition reduction orderings that are total on ground terms are sufficient. The Knuth-Bendix Ordering is widely used for automated theorem proving, because it is well-behaved, i.e. no matter the chosen parameters, syntactically smaller terms tend to be smaller than (significantly) larger terms.

2.2.3. The Superposition Calculus

The superposition calculus is designed to establish whether a set of first-order clauses is unsatisfiable. It is *refutationally complete*, that is, it terminates if the clause set is unsatisfiable, but the calculus may or may not terminate if it is satisfiable. The superposition calculus I present here consist of four inference rules. The calculus operates on a set of clauses and exhaustively applies the inferences until either no inferences are applicable or the empty clause has been derived.

Superposition requires clause-, literal- and term orderings and a selection functions. The selection function, selects a (possibly empty) subset of literals of each clause. Selection overrides the maximality restrictions of the side conditions of the superposition calculus' inferences.

In order to use superposition to prove that a conjecture formula is a consequence of a set of axiom formulas, the conjecture formula must be negated and the negated formula must be added to the axiom set. As described above, the extended set is then transformed to an equisatisfiable set of clauses. If the clause set unsatisfiable, then the conjecture formula follows from the axiom formulas, because its negation is unsatisfiable together with the axioms. If no further inference can be applied and the empty clause has not been derived, the clause set is satisfiable. Each inference has one or two clauses as premise and produce a new clause, which is smaller according to the clause ordering. The calculus allows redundancy elimination, the removal of clauses that are implied by smaller clauses. I present the inference rules and side condition of the superposition calculus adapted from Bachmair and Ganzinger [5].

Positive Superposition
only if conditions 2–8 hold

$$\frac{D' \lor t \approx t' \quad C' \lor s[s_2]_p \approx s'}{(D' \lor C' \lor s[t']_p \approx s')\sigma}\ (PSup)$$

Equality Factoring
only if conditions 1–3 and 10 hold

$$\frac{C' \lor s \approx s' \lor t \approx t'}{(C' \lor t \approx s' \lor t' \not\approx s')\sigma}\ (EF)$$

Negative Superposition
only if conditions 2–6 and 9 hold

$$\frac{D' \lor t \approx t' \quad C' \lor s[s_2]_p \not\approx s'}{(D' \lor C' \lor s[t']_p \not\approx s')\sigma}\ (NSup)$$

Equality Resolution
only if conditions 9 and 11 hold

$$\frac{C' \lor s \not\approx s'}{C'\sigma}\ (ER)$$

Let $\prec$ be a fixed reduction order that is total on ground terms. I refer to $s[s_2]_p$ also as s.

1. σ is the mgu of s and t 2. $s\sigma \not\preceq s'\sigma$ 3. $t\sigma \not\preceq t'\sigma$ 4. s_2 is not a term variable

5. $(t \approx t')\sigma$ is *strictly maximal* in $(D' \lor t \approx t')\sigma$, nothing selected 6. σ is the mgu of t and s_2

7. $(s \approx s')\sigma$ is *strictly maximal* in $(C' \lor s \approx s')\sigma$, nothing selected 8. $t\sigma \approx t'\sigma \not\succeq s\sigma \approx s'\sigma$

9. $((s \not\approx s')\sigma$ is *maximal* in $(C' \lor s \not\approx s')\sigma$, nothing selected$) \lor s \not\approx s'$ selected

10. $(t \approx t')\sigma$ is *maximal* in $(C' \lor s \approx s' \lor t \approx t')\sigma$, nothing selected 11. σ is the mgu of s and s'

Redundancy Redundancy elimination is a critical feature of the superposition calculus. A clause C is *redundant* with respect to a clause set N if it is implied by clauses in N that are smaller than C (according to the clause ordering). Redundancy elimination enables many simplifications such as rewriting, subsumption and others without sacrificing refutational completeness. I will denote the (possibly infinite) set of clauses that is redundant with respect to a clause set N by $Red(N)$. Note that $Red(N)$ may contain clauses not contained in N, i.e. $Red(N)$ is not necessarily a subset of N. A clause set N is *saturated with respect to redundancy* if any clause that can be derived from N with one of the four inference rules is an element of $N \cup Red(N)$.

2.2.4. Refutational Completeness

In this section I will show the proof that if a clause set is unsatisfiable, then the empty clause will be derived, i.e. the proof that the calculus is refutationally complete. The completeness proofs for first-order logic are based on the refutational completeness of the ground case, then extending it to the non-ground case by lifting and model construction. Again, I heavily base the description on the script of the automated reasoning lecture given at Saarland University [38]. The proofs given here are based on a simplification of Waldmann's refutational completeness proof [65, 66] itself an extension of Nieuwenhuis refutational completeness proof for constrained superposition [46] and Bachmair and Ganzinger's refutational completeness proof [5]. I will not introduce the constraints of constrained superposition, but use the same proof layout as Waldmann, i.e. a single induction proof for model construction.

For the remaining section, I need to talk about ground instances of clauses and clause sets so I define the following notation for them. Let C be a clause. Then $G_\Sigma(C)$ is the set of all ground instances of that clause. Let N be a set of clauses. Then $G_\Sigma(N)$ is the set of all ground instances of all clauses in N.

Construction of Ground Candidate Interpretations

The construction of candidate interpretations of saturated sets and the corresponding lemmas follow the description of Bachmair and Ganzinger [5] and Weidenbach [38]:

Let N be a set of clauses not containing the empty clause. Using induction on the clause ordering $\succ_c$, the sets of rewrite rules E_C and R_C for all $C \in G_\Sigma(N)$ are defined as follows: Assume that E_D has already been defined for all $D \in G_\Sigma(N)$ with $D \prec_c C$. Then $R_C = \bigcup_{D \prec_c C} E_D$. The set E_C contains the rewrite rule $s \to t$, if

1. $C = C' \vee s \approx t$,
2. $s \approx t$ is strictly maximal in C,
3. $s \succ t$,
4. C is false in R_C,
5. C' is false in $R_C \cup \{s \to t\}$,
6. s is irreducible with respect to R_C and
7. no negative literal is selected in C'.

In this case, C is called productive. Otherwise $E_C = \emptyset$. Finally, $R_\infty = \bigcup_{D \in G_\Sigma(N)} E_D$.

Lemma 2.2.4.1. If $E_C = \{s \to t\}$ and $E_D = \{u \to v\}$, then $s \succ u$ if and only if $C \succ_c D$.

Lemma 2.2.4.2. The rewrite systems R_C and R_∞ are convergent (i.e. confluent and terminating).

Proof. Obviously, $s \succ t$ for all rules $s \to t$ in R_C and R_∞. Furthermore, it is easy to check that there are no critical pairs between any two rules: Assume that there are rules $u \to v$ in E_D and $s \to t$ in E_C such that u is a subterm of s. As $\succ$ is a reduction ordering that is total on ground terms, I get $u \prec s$ and therefore $D \prec_c C$ and $E_D \subseteq R_C$. But then s would be reducible by R_C, contradicting condition (6). □

Corollary 2.2.4.3. If $D \in G_\Sigma(N)$ is true in R_D, then D is true in R_∞ and R_C for all $C \succ_c D$.

Corollary 2.2.4.4. If $D = D' \vee u \approx v$ is productive, then D' is false and D is true in R_∞ and R_C for all $C \succ_c D$.

Lifting to Variables

with the main differences that I extracted Lemma 2.2.4.5 from Lemma 2.2.4.8, which it was previously part of. This makes the proofs in the typed cases simpler, because Lemma 2.2.4.8 can remain unmodified. The lemma can remain unmodified, because all its assumptions are extracted into individual lemmas. Then only the lemmas for the assumptions, need to be shown again for the typed case.

For the refutational completeness proof to succeed, two main assumptions must hold for variables. First, variable instances must be closed under rewriting with $R_{C\theta}$, i.e. it must be possible to instantiate a variable to all terms that its instances can be rewritten to. Second, inferences from ground-substituted clauses must be instances of inferences of the non-ground clauses. These properties are expressed in the following three lemmas. With the help of these three lemmas, the remaining proof of the Model Construction Lemma is independent of how variables are actually defined, i.e. if they are untyped or restricted by types.

Lemma 2.2.4.5 (Variable instances are closed under rewriting with $R_{C\theta}$)**.**
Let $C \in N$ and θ be a substitution such that $C\theta \in G_\Sigma(C)$ and u be a variable occurring in C. If $u\theta \to_{R_{C\theta}} t$ then there exists a θ' such that $u\theta' = t$ and $C\theta' \in G_\Sigma(C)$.

Lemma 2.2.4.6 (Lifting Lemma for the equality resolution and equality factoring inferences [38])**.**

Let C be a clause and let θ be a substitution such that $C\theta$ is ground. Then every equality resolution (or equality factoring) inference of $C\theta$ is an instance of an equality resolution (or equality factoring) inference from C.

Lemma 2.2.4.7 (Lifting Lemma for the superposition inferences [38])**.**
Let $C = C' \vee s \approx s'$ and $D = D' \vee t \approx t'$ be two clauses without common variables and let θ be a substitution such that $C\theta$ and $D\theta$ are ground. If there is a superposition inference between $C\theta$ and $D\theta$ where $s\theta$ and some subterm of $t\theta$ are overlapped and $s\theta$ does not occur in $t\theta$ at or below a variable position of t, then the inference is an instance of a superposition inference from C and D.

Model Construction

The model construction shows that any saturated (with respect to redundancy) set of clauses that does not contain the empty clause has a term-generated model.

Theorem 2.2.4.8 (Model Construction [38])**.**
Let N be a set of clauses that is saturated up to redundancy and does not contain the empty clause. Then for every ground clause $C\theta \in G_\Sigma(N)$ it holds that:

1. $E_{C\theta} = \emptyset$ if and only if $C\theta$ is true in $R_{C\theta}$.
2. If $C\theta$ is redundant with respect to $G_\Sigma(N)$, then it is true in $R_{C\theta}$.
3. $C\theta$ is true in R_∞ and in R_D for every $D \in G_\Sigma(N)$ with $D \succ_C C\theta$

Proof. The following proof of the theorem does not consider selection. It uses case distinction and induction on the clause ordering $\succ_C$ and assume that 1–3 are already satisfied for all clauses in $G_\Sigma(N)$ that are smaller than $C\theta$. Note that the "if" part of condition 1 is obvious from the construction and that condition 3 follows immediately from condition 1 and Corollaries 2.2.4.3 and 2.2.4.4. So it remains to show condition 2 and the "only if" part of condition 1.

1. $C\theta$ is redundant with respect to $G_\Sigma(N)$.
   If $C\theta$ is redundant with respect to $G_\Sigma(N)$, then it follows from clauses in $G_\Sigma(N)$ that are smaller than $C\theta$. By condition 3 of the induction hypothesis, these clauses are true in $R_{C\theta}$. Hence $C\theta$ is true in $R_{C\theta}$.

2. $x\theta$ is reducible by $R_{C\theta}$.
   Suppose there is a variable x occurring in C such that $x\theta$ is reducible by $R_{C\theta}$, say $x\theta \rightarrow_{R_{C\theta}} t$. Since variables are closed under rewriting with $R_{C\theta}$ (Lemma 2.2.4.5), there exists a substitution θ' such that $x\theta' = t$ and $y\theta' = y\theta$ for every variable $y \neq x$. The clause $C\theta'$ is smaller than $C\theta$. By condition 3 of the induction hypothesis, it is true in $R_{C\theta}$. By congruence, every literal of $C\theta$ is true in $R_{C\theta}$ if and only if the corresponding literal of $C\theta'$ is true in $R_{C\theta}$; hence $C\theta$ is true in $R_{C\theta}$.

3. $C\theta$ contains a maximal negative literal.
   Suppose that $C\theta$ does not fall into Case 1 or 2 and that $C\theta = C'\theta \vee s\theta \not\approx s'\theta$, where $s\theta \not\approx s'\theta$ is maximal in $C\theta$. If $s\theta \approx s'\theta$ is false in $R_{C\theta}$, then $C\theta$ is clearly true in $R_{C\theta}$ and we are done. So assume that $s\theta \approx s'\theta$ is true in $R_{C\theta}$, that is, $s\theta \downarrow_{R_{C\theta}} s'\theta$. Without loss of generality, assume that $s\theta \succeq s'\theta$.

   a) $s\theta = s'\theta$.
      If $s\theta = s'\theta$, then there is an equality resolution inference

$$\frac{C'\theta \vee s\theta \not\approx s'\theta}{C'\theta}$$

      As shown in the Lifting Lemma (Lemma 2.2.4.6), this is an instance of an equality resolution inference

$$\frac{C' \vee s \not\approx s'}{C'\sigma}$$

      where $C = C' \vee s \not\approx s'$ is contained in N and σ is the most general unifier of s and s'. Since $C\theta$ is not redundant with respect to $G_\Sigma(N)$, C is not redundant with respect to

N. As N is saturated up to redundancy, the conclusion $C'\sigma$ of the inference from C is contained in N or is redundant with respect to N. Therefore, $C'\theta$ is either contained in $G_\Sigma(N)$ and smaller than $C\theta$, or it follows from clauses in $G_\Sigma(N)$ that are smaller than itself (and therefore smaller than $C\theta$). By the induction hypothesis, clauses in $G_\Sigma(N)$ that are smaller than $C\theta$ are true in $R_{C\theta}$, thus $C'\theta$ and $C\theta$ are true in $R_{C\theta}$.

b) $s\theta \succ s'\theta$.
Without loss of generality assume that C and D are variable disjoint; so the same substitution θ can be used. If $s\theta \downarrow_{R_{C\theta}} s'\theta$ and $s\theta \succ s'\theta$, then $s\theta$ must be reducible by some rule in some $E_{D\theta} \subseteq R_{C\theta}$. Let $D\theta = D'\theta \vee t\theta \approx t'\theta$ with $E_{D\theta} = \{t\theta \to t'\theta\}$. Since $D\theta$ is productive, $D'\theta$ is false in $R_{C\theta}$. Besides, by condition 2 of the induction hypothesis, $D\theta$ is not redundant with respect to $G_\Sigma(N)$, so D is not redundant with respect to N. Note that $t\theta$ cannot occur in $s\theta$ at or below a variable position of s since otherwise $C\theta$ would be subject to Case 2 above. Thus, the left superposition inference

$$\frac{D'\theta \vee t\theta \approx t'\theta \quad C'\theta \vee s\theta[t\theta] \not\approx s'\theta}{D'\theta \vee C'\theta \vee s\theta[t'\theta] \not\approx s'\theta}$$

can be applied. From the Lifting Lemma (Lemma 2.2.4.7) it follows that this is a ground instance of a left superposition inference from D and C. By saturation up to redundancy, its conclusion is either contained in $G_\Sigma(N)$ and smaller than $C\theta$, or it follows from clauses in $G_\Sigma(N)$ that are smaller than itself and therefore are smaller than $C\theta$. By the induction hypothesis, these clauses are true in $R_{C\theta}$, thus $D'\theta \vee C'\theta \vee s\theta[t'\theta] \approx s'\theta$ is true in $R_{C\theta}$. Since $D'\theta$ and $s\theta[t'\theta] \approx s'\theta$ are false in $R_{C\theta}$, both $C'\theta$ and $C\theta$ must be true.

4. $C\theta$ does not contain a maximal negative literal.
Suppose that $C\theta$ does not fall into Cases 1 to 3. Then $C\theta$ can be written as $C'\theta \vee s\theta \approx s'\theta$, where $s\theta \approx s'\theta$ is a maximal literal of $C\theta$. If $E_{C\theta} = \{s\theta \to s'\theta\}$ or $C'\theta$ is true in $R_{C\theta}$ or $s\theta = s'\theta$, then there is nothing to show. It remains to show what happens in the case when $E_{C\theta} = \emptyset$ and $C'\theta$ is false in $R_{C\theta}$. Without loss of generality assume that $s\theta \succ s'\theta$.

a) $s\theta \approx s'\theta$ is maximal in $C\theta$, but not strictly maximal.
If $s\theta \approx s'\theta$ is maximal in $C\theta$, but not strictly maximal, then $C\theta$ can be written as $C''\theta \vee t\theta \approx t'\theta \vee s\theta \approx s'\theta$, where $t\theta = s\theta$ and $t'\theta = s'\theta$. Thus, there is a equality factoring inference

$$\frac{C''\theta \vee t\theta \approx t'\theta \vee s\theta \approx s'\theta}{C''\theta \vee t'\theta \not\approx s'\theta \vee t\theta \approx t'\theta}$$

From the Lifting Lemma (Lemma 2.2.4.6) it follows that this inference is a ground instance of an inference from C. The conclusion is smaller since the negative literal contains the smaller term and the positive literal stays unchanged. Thus, by the induction hypothesis we known that the conclusion is true in $R_{C\theta}$. $C''\theta$ must be false (or $C\theta$ would be true) and $t'\theta = s'\theta$ implies that $t'\theta \not\approx s'\theta$ and thus $C''\theta \vee t'\theta \not\approx s'\theta$ must be also false. For the conclusion to be true, $t\theta \approx t'\theta$ must be true in $R_{C\theta}$ and thus $s\theta \approx s'\theta$ (and therefore $C\theta$) must be true.

b) $s\theta \approx s'\theta$ is strictly maximal in $C\theta$ and $s\theta$ is reducible.
Suppose that $s\theta \approx s'\theta$ is strictly maximal in $C\theta$ and $s\theta$ is reducible by some rule in $E_{D\theta} \subseteq R_{C\theta}$. Let $D\theta = D'\theta \vee t\theta \approx t'\theta$ and $E_{D\theta} = \{t\theta \to t'\theta\}$. Since $D\theta$ is productive, $D\theta$ is not redundant and $D'\theta$ is false in $R_{C\theta}$. We can now proceed in essentially the same way as in Case 3b: We know that $t\theta$ does not occurred in $s\theta$ at or below a variable position of s, because then $C\theta$ would be subject to Case 2 above. Thus, the right superposition inference

$$\frac{D'\theta \vee t\theta \approx t'\theta \quad C'\theta \vee s\theta[t\theta] \approx s'\theta}{D'\theta \vee C'\theta \vee s\theta[t'\theta] \approx s'\theta}$$

applies. From the Lifting Lemma (Lemma 2.2.4.7) it follows that this is a ground instance of a superposition inference from D and C. Since $t\theta \succ t'\theta$ (otherwise there would be no rewrite rule) we know that $s\theta[t'\theta] \approx s'\theta$ is smaller then $s\theta[t\theta] \approx s'\theta$. Since $s\theta[t\theta] \approx s'\theta$ is strictly maximal, we also know that it is larger than all literals in $C'\theta$. It is also larger than all literals in $D'\theta$. and thus the conclusion is smaller than $C\theta$. By saturation up to redundancy and the induction hypothesis we know that the conclusion is true in $R_{C\theta}$. Since $D'\theta$ and $C'\theta$ are false in $R_{C\theta}$, $s\theta[t'\theta] \approx s'\theta$ must be true in $R_{C\theta}$. On the other hand, $t\theta \approx t'\theta$ is true in $R_{C\theta}$, so by congruence, $s\theta[t\theta] \approx s'\theta$ and $C\theta$ are true in $R_{C\theta}$.

c) $s\theta \approx s'\theta$ is strictly maximal in $C\theta$ and $s\theta$ is irreducible.
Suppose that $s\theta \approx s'\theta$ is strictly maximal in $C\theta$ and $s\theta$ is irreducible by $R_{C\theta}$. Then it must be one of three possibilities:

i. $C\theta$ can be true in $R_{C\theta}$.
ii. $E_{C\theta} = \{s\theta \to s'\theta\}$.
iii. $C'\theta$ can be true in $R_{C\theta} \cup \{s\theta \to s'\theta\}$.

In the first two cases there is nothing to show. Let us therefore assume that $C\theta$ is false in $R_{C\theta}$ and $C'\theta$ is true in $R_{C\theta} \cup \{s\theta \to s'\theta\}$. Then $C'\theta = C''\theta \vee t\theta \approx t'\theta$, where the literal $t\theta \approx t'\theta$ is true in $R_{C\theta} \cup \{s\theta \to s'\theta\}$ and false in $R_{C\theta}$. In other words, $t\theta \downarrow_{R_{C\theta} \cup s\theta \to s'\theta} t'\theta$, but not $t\theta \downarrow_{R_{C\theta}} t'\theta$. Consequently, there is a rewrite proof of $t\theta \to^* u \leftarrow^* t'\theta$ by $R_{C\theta} \cup \{s\theta \to s'\theta\}$ in which the rule $s\theta \to s'\theta$ is used at least once. Without loss of generality, assume that $t\theta \succeq t'\theta$. Since $s\theta \approx s'\theta \succ_L t\theta \approx t'\theta$ and $s\theta \succ s'\theta$ I can conclude that $s\theta \succeq t\theta \succ t'\theta$. Then there is only one possibility how the rule $s\theta \to s'\theta$ can be used in the rewrite proof: $s\theta = t\theta$ must hold and the rewrite proof must have the form $t\theta \to s'\theta \to^* u \leftarrow^* t'\theta$, where the first step uses $s\theta \to s'\theta$ and all other steps use rules from $R_{C\theta}$. Consequently, $s'\theta \approx t'\theta$ is true in $R_{C\theta}$. Then there is an equality factoring inference

$$\frac{C''\theta \vee t\theta \approx t'\theta \vee s\theta \approx s'\theta}{C''\theta \vee t'\theta \not\approx s'\theta \vee t\theta \approx t'\theta}$$

From the Lifting Lemma (Lemma 2.2.4.6) we know that it is a ground instance of an equality factoring inference of C. The conclusion is smaller because $s\theta \succeq t\theta \succ t'\theta$ and $s\theta \succ s'\theta$. The conclusion of the ground inference is true in $R_{C\theta}$ since the clause set it saturated up to redundancy and because of the induction hypothesis. Since the literal $t'\theta \approx s'\theta$ must be false in $R_{C\theta}$, the rest of the clause must be true in $R_{C\theta}$, and therefore $C\theta$ must be true in $R_{C\theta}$.

□

Refutational Completeness

Now, the actual refutational completeness theorems for superposition remain. Refutational completeness exists in two flavors. The first flavor is *static refutational completeness*, which means that a saturated clause set has a model if and only if it does not contain the empty clause. The second flavor is *dynamic refutational completeness*, which means that for any unsatisfiable clause set, under some fairness restrictions, any sequence of inference steps eventually derives the empty clause. Again, I follow the presentation from Weidenbach [38].

The Static View

For the static view on completeness, the model constructed by the model construction has to be lifted from the ground clauses to the non-ground clauses. Then static refutational completeness is a consequence of the Model Construction Lemma.

Lemma 2.2.4.9 (Model Lifting [38])**.**
Let N be a set of (universally quantified) Σ-clauses and let $\mathcal{I}$ be a term-generated Σ-interpretation. Then $\mathcal{I}$ is a model of $G_\Sigma(N)$ then it is a model of N.

Theorem 2.2.4.10 (Herbrand [38])**.**
A countable set N of first-order clauses is satisfiable if and only if it has a term-generated model.

Proof. $\Leftarrow$ If it has a term-generated model it is satisfiable
$\Rightarrow$ Let N be satisfiable and thus $N \not\models \bot$.

$$\begin{array}{lll} N \not\models \bot & \Rightarrow \bot \notin \mathrm{Sup}^*(N) & \text{Superposition is sound} \\ & \Rightarrow \bot \notin G_\Sigma(\mathrm{Sup}^*(N)) & \\ & \Rightarrow G_\Sigma(\mathrm{Sup}^*(N))_\mathcal{I} \models G_\Sigma(\mathrm{Sup}^*(N)) & \text{(Lemma 2.2.4.8 with 2.2.4.5, 2.2.4.6 and 2.2.4.7)} \\ & \Rightarrow G_\Sigma(\mathrm{Sup}^*(N))_\mathcal{I} \models \mathrm{Sup}^*(N) & \text{(Lemma 2.2.4.9)} \\ & \Rightarrow G_\Sigma(\mathrm{Sup}^*(N))_\mathcal{I} \models N & (N \subseteq \mathrm{Sup}^*(N)) \end{array}$$

□

Theorem 2.2.4.11 (Static Refutational Completeness [38])**.**
Let N be a set of clauses that is saturated up to redundancy. Then N has a model if and only if N does not contain the empty clause.

Proof. If $\bot \in N$, then obviously N does not have a model. If $\bot \notin N$, then the interpretation R_∞ is a model of all ground instances in of N according to Part 3. of the Model Construction Lemma. As R_∞ is term-generated, it is a model of $G_\Sigma(N)$. Thus, by Herbrand's Theorem it is also a model of N (Theorem 2.2.4.10). □

The Dynamic View

For the dynamic view on completeness, fairness needs to be defined in such a way that the limit of any fair run is saturated (up to redundancy). Then dynamic refutational completeness can be shown using fairness and static refutational completeness.

An inference is *enabled* with respect to a clause set N if all its premises are contained in N and its result is not contained in N or $Red(N)$. A *run* of the superposition calculus is a sequence of clause sets $N_0 \vdash N_1 \vdash N_2 \vdash \cdots$, such that $N_i \models N_{i+1}$ and all clauses in $N_i \backslash N_{i+1}$ are redundant with respect to N_{i+1}. Each step takes the previous clause set adds any number of clauses that follow from that clause set and removes any number of clauses that follow from smaller clauses of the new clause set. A run is *fair* if for all i and inferences sup that are enabled at N_i there exists a j with $j > i$ such that the inference sup is not enabled in N_j.

Lemma 2.2.4.12 (Limit is saturated [38])**.**
If a run is fair, then its limit is saturated up to redundancy.

Proof. If the limit is not saturated up to redundancy, there must be an inference that can still be applied. Thus both its premises must be in N_* but the result is not in N_* and is not redundant with respect to N_*. Such an inference must be enabled in N_* and thus must be enabled in all j that are $j \geq i$ for some i. By definition such a run cannot be fair. □

Lemma 2.2.4.13 (Redundant Clauses [38])**.**
Let C be a clause that is redundant with respect to a clause set N and let $\mathcal{I}$ be an interpretation such that $\mathcal{I} \models N$ then $\mathcal{I} \models C$.

Theorem 2.2.4.14 (Dynamic Refutational Completeness [38])**.**
Let $N_0 \vdash N_1 \vdash N_2 \ldots$ be a fair run and N_* its limit. Then N_0 has a model if and only if $\bot \notin N_*$.

Proof. $\Rightarrow$ Obvious.
$\Leftarrow$ The run is fair and therefore N_* is saturated up to redundancy (Lemma 2.2.4.12). If $\bot \notin N_*$ and since we know that N_* is saturated up to redundancy, it has to have a model (Theorem 2.2.4.11). Every clause in N_0 is an element of N_* or is redundant with respect to N_* and thus the model of N_* is also a model of N_0 (Lemma 2.2.4.13). □

3. Typed Superposition

In this chapter I present how to extend the superposition calculus with a polymorphic type system that supports type classes. The type system is designed to be a first-order version of the type system used by the interactive and higher-order logic proof assistant Isabelle/HOL [49]. In particular, this typed version of superposition enables translations of proof obligations from Isabelle without any type-encoding related overhead. Encoding of other features, such as higher-order features like higher-order quantification (the superposition calculus is still first-order), partial applications and lambdas is still required.

To achieve this, I first define a first-order logic extended with such a type system (Sect. 3.1). Afterwards, I show how to adapt the superposition calculus and its machinery to this typed setting. The main changes to superposition that are required to include a type system are contained within unification (Sect. 3.1.3), Skolemization (Sect. 3.1.5) and the orderings (Sect. 3.2). The inference rules of the superposition calculus itself remains largely unchanged (Sect. 3.3).

To restrict the search space, I have made the decision to not consider combinations of type classes that do not contain any ground types, i.e. to consider those combinations of type classes to be invalid. This has the benefit of limiting the reasoning to the relevant parts of the type classes (and their combinations). A side effect is that type-Skolemization for those combinations of type classes (without ground types) is either impossible or not satisfiability preserving (Sect. 3.1.5).

An evaluation of an implementation of the presented typed superposition calculus against type-encodings used by Isabelle and a monomorphic type system provides evidence to the type systems usefulness (Sect. 3.4). The benchmark problems used for the evaluation are generated by Sledgehammer [10] from Isabelle formalizations. The formalizations that I used are either included in Isabelle or from the Archive of Formal Proofs (AFP, www.isa-afp.org).

Furthermore, I show refutational completeness of the superposition calculus extended with the type system by lifting untyped ground superposition via intermediate type-symbol and monomorphic first-order languages to the polymorphic type system presented below (Sect. 3.5).

Finally, I discuss related work (Sect. 3.6).

3.1. Polymorphic First-Order Logic with Type Classes

In this section, I define the first-order language that includes the polymorphic type system extended with type classes. The language features terms consisting of variables and functions with complex types. These complex types allow type terms consisting of type constructors and type variables. The type variables are further restricted by (sets of) type classes to only range over a certain subset of all available types.

As before, in the untyped first-order logic (presented in section 2.1.1), I first define the syntax (Sect. 3.1.1). In contrast to the untyped first-order setting, I also have to define typing rules that determine if terms are well-typed. The description of the typing rules is therefore presented directly after the syntax (Sect. 3.1.2). Following this, I present unification for the typed terms (Sect. 3.1.3). In that section, I also introduce an alternative representation of the typed terms that makes unifications easier and faster to compute. Then, I define the semantics of the typed language and finally show the typed clausification (Sect. 3.1.4) and discuss clausification and Skolemization (Sect. 3.1.5).

3.1.1. Syntax

I now present the syntax of the polymorphic language with type classes. First, I present the signature followed by the types, terms and formulas. In contrast to the untyped case more information must be encoded in the signature to express the type of each symbol. Terms also must be able to express the types. There is also additional structures to represent type terms, type classes and type-class constraints, which restrict type variables.

Definition 3.1.1.1 (**Signature**). A signature for a polymorphic first-order language with type classes is a tuple $\Sigma = (S_\mathcal{F}, S_\mathcal{P}, S_\mathcal{T}, S_\mathcal{K}, \mathcal{TC}, \mathcal{T}, \mathcal{F}, \mathcal{P})$ where

$S_\mathcal{F}$	is the set of function symbols,
$S_\mathcal{P}$	is the set of predicate symbols,
$S_\mathcal{T}$	is the set of type constructor symbols,
$S_\mathcal{K}$	is the set of type class symbols,
$\mathcal{TC}$	is the set of subclass declarations,
$\mathcal{T}$	is the set of type declarations,
$\mathcal{F}$	is the set of function declarations and
$\mathcal{P}$	is the set of predicate declarations.

Every symbol has a fixed arity. There must be exactly one declaration per function and predicate symbol.

Before I formally define $\mathcal{T}$, $\mathcal{F}$ and $\mathcal{P}$ I need the definitions of (possibly not well-formed) type-class constraints, type terms, terms and formulas. Section 3.1.2 then introduces rules that determine if they are well-formed. In particular, those rules prevent formulas with non-prenex type quantifiers and ill-typed terms.

Notation I use f for function symbols, p for predicate symbols, t, s for terms, ϕ, ψ for formulas, u, v for term variables, α for type variables, σ, θ for substitutions, τ for type terms, κ for type constructors, k for type classes, K for sets of type classes (i.e. type-class constraints), n for type arity and m for term arity. As before, I write $t[s]_p$ to denote that the term s is placed at position p in another term t. If not explicitly mentioned otherwise, *variables* means both type and term variables.

Subclass Declarations The subclass declarations are defined as follows:

$\mathcal{TC}$ is a finite set of elements $k_1 \subseteq k_2 \in \mathcal{TC}$, which are subclass declarations, i.e. k_1 is a subclass of k_2. The structure of $\mathcal{TC}$ must form a directed acyclic graph (DAG).

Let $\subseteq^*_{\mathcal{TC}}$ be the reflexive, transitive closure of the subclass declarations $k_1 \subseteq k_2 \in \mathcal{TC}$.

Definition 3.1.1.2 (Type-class constraint). A *type-class constraint* $K \subseteq S_{\mathcal{K}}$ is a set of type-class symbols.

Each type class represents a set of types and the type-class constraint represents the intersection of all the type sets of its type classes. For a type constraints K, I also write $K = k_1 \,\&\, \ldots \,\&\, k_n$ instead of the set representation $K = \{k_1, \ldots, k_n\}$. I assume in the remaining chapter, that the type-class constraints are *minimal*, i.e. that they contain no type class that has a subclass in the type-class constraint. A type class is automatically fulfilled if one of its subclasses is and can therefore be removed if a subclass is present.

Definition 3.1.1.3 (Type terms). Let $\mathcal{X}_\tau$ be a given countably infinite set of type variables. All possibly not well-formed *type terms* are recursively defined as:

- Every type variable in $\mathcal{X}_\tau$ is a type term
- For all $\kappa \in S_{\mathcal{T}}$, if $\tau_1, \ldots, \tau_n$ are type terms, then $\kappa(\tau_1, \ldots, \tau_n)$ is a type term.

Type terms that do not contain a type variable are *monomorphic*.

Definition 3.1.1.4 (Terms). Let $\mathcal{X}_t$ be a given countably infinite set of term variables. All possibly not well-formed *terms* are recursively defined as:

- Every term variable in $\mathcal{X}_t$ is a term
- For all $f \in S_{\mathcal{F}}$ if $\tau_1, \ldots, \tau_n$ are type terms and $t_1, \ldots, t_m$ are terms, then $f[\tau_1, \ldots, \tau_n](t_1, \ldots, t_m)$ is also a term.

Terms that do not contain a term variable are *ground* and those that do not contain a type variable are *monomorphic*. Terms that have the form $f[\tau_1, \ldots, \tau_n](t_1, \ldots, t_m)$ are *function terms*.

Definition 3.1.1.5 (Formulas). All possibly not well-formed *formulas* are recursively defined as:

1. $\bot$ is a formula
2. $\top$ is a formula
3. For all $p \in S_{\mathcal{P}}$ if $\tau_1, \ldots, \tau_n$ are type terms and $t_1, \ldots, t_m$ are terms, then $p[\tau_1, \ldots, \tau_n](t_1, \ldots, t_m)$ is a formula.
4. If t_1, t_2 are terms, then $t_1 \approx t_2$ is a formula.
5. If ϕ_1, ϕ_2 are formulas, then $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \Rightarrow \phi_2$, $\phi_1 \Leftrightarrow \phi_2$ and $\neg\phi_1$ are also formulas.
6. If ϕ is a formula, u a term variable and τ a type term, then $\forall u{:}\tau.\ \phi$ and $\exists u{:}\tau.\ \phi$ are also formulas. Within ϕ the term variable u is *bound* and of type τ (unless it is bound again in subformulas of ϕ. Then it is not bound in those subformulas, but the u of the subformula is bound).

7. If ϕ is a formula, α a type variable and K a type-class constraint, then $\forall_\tau \alpha{:}K.\, \phi$ and $\exists_\tau \alpha{:}K.\, \phi$ are also formulas. Within ϕ the type variable α is *bound* and of type-class constraint K (unless it is bound again in subformulas of ϕ, then it is not bound in those subformulas, but the α of the subformula is bound).

Note that the quantifiers in 6 range over term variables and the quantifiers in 7 ranges over type variables. In formulas, type quantifiers never occur below other formula symbols (except other type quantifiers). The formulas 1–4 are considered *atomic formulas*. Atomic formulas and their negations ($\neg$) are *literals*. A *closed formula* is a formula in which every variable is bound by a quantifier. I only consider closed formulas. Without loss of generality, I assume that every variable has a unique name. In closed formulas with unique variable names, there is always a unique binding quantifier for each variable and thus it is always known which type a term variable has and which type-class constraint a type variable has.

Definition 3.1.1.6 (Clause). A *clause* is a disjunction of literals. All variables of a clause are implicitly universally quantified and thus implicitly given their type and type-class constraint.

The clause consisting of no literals is called $\bot$ or the *empty clause*. A clause can be written as a formula (then with explicit quantification). Thus, if I have definitions or proofs for formulas, I do not duplicate them for clauses. Formulas and clauses are *(term) ground* if they contain no (term) variables.

Now that I have defined possibly not well-formed (type) terms and formulas, I can properly define the remaining elements of the signature. They are:

$\mathcal{T}$ is a finite set of elements $\forall_\tau\ \alpha_1{:}K_1,\ \ldots,\ \alpha_n{:}K_n.\ \kappa(\alpha_i,\ \ldots,\ \alpha_m) : k \in \mathcal{T}$, which are type declarations. The "$: k$" part is optional. If omitted, the type constructor κ does not belong to any type class and there must be exactly one declaration for κ. For every type constructor κ in $S_\mathcal{T}$ there must be at least one declaration.

$\mathcal{F}$ is a finite set of elements $f : \forall_\tau\ \alpha_1{:}K_1,\ \ldots,\ \alpha_n{:}K_n.\ \tau_1,\ \ldots,\ \tau_m \to \tau \in \mathcal{F}$ which are function declarations. The type variables $\alpha_1,\ \ldots,\ \alpha_n$ must be the only type variables occurring in the type terms $\tau_1,\ \ldots,\ \tau_m$ and τ. I use $f[\tau'_1,\ \ldots,\ \tau'_n](t_1,\ \ldots,\ t_m)$, to write a function term, denoting the appropriate instantiations of the α_i by the corresponding type argument τ'_i. The type terms of the declaration can then be recovered by applying the substitution $\{\alpha_i \mapsto \tau'_1,\ \ldots,\ \alpha_n \mapsto \tau'_n\}$ to the $\tau_1,\ \ldots,\ \tau_m$ and the τ.

$\mathcal{P}$ is the finite set of elements $p : \forall_\tau\ \alpha_1{:}K_1,\ \ldots,\ \alpha_n{:}K_n.\ \tau_1,\ \ldots,\ \tau_m \in \mathcal{P}$, which are predicate declarations. The type variables $\alpha_1,\ \ldots,\ \alpha_n$ must be the only type variables occurring in the type terms $\tau_1,\ \ldots,\ \tau_m$ and τ. Similar to functions I write $p[\tau'_1,\ \ldots,\ \tau'_n](t_1,\ \ldots,\ t_m)$, denoting the appropriate instantiations.

A type class constraint K is more general than a type class constraint K' ($K' \leq_{\mathcal{TC}} K$) if and only if for all $k \in K$ there is a $k' \in K'$ such that $k' \subseteq^*_{\mathcal{TC}} k$.

Definition 3.1.1.7 (Standard Coregularity, adapted from [26, 56]**).**
The set of type declarations $\mathcal{T}$ is *coregular'* if for all $\forall_\tau\ \alpha_1{:}K_1,\ \ldots,\ \alpha_n{:}K_n.\ \kappa(\alpha_i,\ \ldots,\ \alpha_m) : k \in \mathcal{T}$ and $\forall_\tau\ \alpha_1{:}K'_1,\ \ldots,\ \alpha_n{:}K'_n.\ \kappa(\alpha_i,\ \ldots,\ \alpha_m) : k' \in \mathcal{T}$ such that $k' \subseteq^*_{\mathcal{TC}} k$, it holds that $K'_i \leq_{\mathcal{TC}} K_i$ for all i.

Coregularity ensures that there are most general unifiers for order-sorted algebras [48, 56]. Since $\mathcal{T}$ does not require a type declaration for all combinations of type constructors and type classes, coregularity has to take that into account by also considering the type declarations that the type classes 'inherit' (via the subclass relation) from their subclasses.

Definition 3.1.1.8 (**Coregularity**, further adapted from [26, 56]**).**
The set of type declarations $\mathcal{T}$ is *coregular* if for all $\forall_\tau\ \alpha_1{:}K_1, \ldots, \alpha_n{:}K_n.\ \kappa(\alpha_i, \ldots, \alpha_m) : k \in \mathcal{T}$ and $\forall_\tau\ \alpha_1{:}K_1', \ldots, \alpha_n{:}K_n'.\ \kappa(\alpha_i, \ldots, \alpha_m) : k' \in \mathcal{T}$ such that $k' \subseteq^*_{\mathcal{TC}} k$, it holds that $K_i' \leq_{\mathcal{TC}} K_i$ for all i. Furthermore, for all type classes k and type constructors κ, there exists a type class $k' \subseteq^*_{\mathcal{TC}} k$ with $\forall_\tau\ \alpha_1{:}K_1', \ldots, \alpha_n{:}K_n'.\ \kappa(\alpha_i, \ldots, \alpha_m) : k' \in \mathcal{T}$ such that for all $k'' \subseteq^*_{\mathcal{TC}} k$ with $\forall_\tau\ \alpha_1{:}K_1'', \ldots, \alpha_n{:}K_n''.$ $\kappa(\alpha_i, \ldots, \alpha_m) : k'' \in \mathcal{T}$ such that $K_i'' \leq_{\mathcal{TC}} K_i'$ for all i.

Note, if a type class k has a type declaration for a type constructor κ, the new constraint is fulfilled if the original coregularity was fulfilled, because then k is the k' we look for. The new constraint adds the check for the subclass from which the type class 'inherits' its type declaration.

I require that $\mathcal{T}$ is coregular. Isabelle's constructive type classes are also coregular [26, 48].

3.1.2. Typing Rules

So far, I have only defined formulas, terms and type terms with no restriction as to whether they are well-typed. The main objective of the typing rules is to ensure that declared and used types of terms match with the declarations and quantors, to prevent that a type quantifier occurs below a term quantifier and to define which type classes a type-term belongs to. I now define the typing rules for formulas, terms and type terms.

Let $\gamma^{\mathcal{T}}_{\mathcal{C}}$ represent a typing context, where $\mathcal{T}$ is a mapping from term variables to types and $\mathcal{C}$ is a mapping from type variables to a set of type classes (type-class constraint). All well-typedness assertions are implicitly relative to a given signature. Without loss of generality, I assume that all variables are named uniquely. I further assume that $o_{\mathcal{B}}$, o are boolean pseudo types which do not match any type in the signature. I use these two boolean types to fix the quantifier for types ($\forall_\tau$, $\exists_\tau$) to appear only on the very top position in formulas, so that formulas of the type $o_{\mathcal{B}}$ can only be extended by further type quantifiers.

I will give the typing rules in form of inference rules, which follow the following structure: The elements below the bar of the rules are what we want to show (the conclusion). The elements above the bar are the assumptions and have to be proven.

A closed formula ϕ is *well-typed* if and only if one can, starting from $\gamma^{\emptyset}_{\emptyset} \vdash \phi : o$, apply the inference rules until all assumptions are empty or follow directly from the signature (i.e. the assumptions are checks if a declaration is an element of the corresponding set in the signature).

Formulas The typing rules for formulas have two main objectives. First, to ensure that the type quantifiers ($\forall_\tau$, $\exists_\tau$) are always at the top of the formulas to prevent a type quantifier to be below a term quantifier. Secondly, that the arguments of predicates match the predicate's definition and are well-typed, and that the type of both terms of an equality is the same.

Definition 3.1.2.1 (Formula Typing Rules). The typing rules for formulas are as follows:

$$\frac{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash s : \tau \quad \gamma_{\mathcal{C}}^{\mathcal{T}} \vdash t : \tau}{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash s \approx t : o_{\mathcal{B}}}\ (\approx) \qquad \frac{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash \phi_1 : o_{\mathcal{B}} \quad \gamma_{\mathcal{C}}^{\mathcal{T}} \vdash \phi_2 : o_{\mathcal{B}}}{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash \phi_1 \# \phi_2 : o_{\mathcal{B}}}\ (\# {\in} \{\wedge,\vee,\Rightarrow,\Leftrightarrow\}) \qquad \frac{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash \phi : o_{\mathcal{B}}}{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash \neg\phi : o_{\mathcal{B}}}\ (\neg)$$

$$\frac{p : \forall \alpha_1{:}K_1, \ldots, \alpha_n{:}K_n.\ \tau_1, \ldots, \tau_m \in \mathcal{P} \quad \gamma_{\mathcal{C}}^{\mathcal{T}} \vdash t_1 : \tau_1\sigma \quad \ldots \quad \gamma_{\mathcal{C}}^{\mathcal{T}} \vdash t_n : \tau_n\sigma}{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash p[\alpha_1\sigma, \ldots, \alpha_n\sigma](t_1, \ldots, t_n) : o_{\mathcal{B}}}\ (\text{pred})$$

and for all $i : 1..m.\ \mathcal{C} \vdash \alpha_i\sigma\ \mathbf{:}\ K_i$

$$\frac{\gamma_{\mathcal{C}}^{\mathcal{T}[u\mapsto\tau]} \vdash \phi : o_{\mathcal{B}}}{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash \# u : \tau.\ \phi : o_{\mathcal{B}}}\ (\# {\in} \{\forall,\exists\}) \qquad \frac{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash \phi : o_{\mathcal{B}}}{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash \phi : o}\ (\text{top}) \qquad \frac{\gamma_{\mathcal{C}[\alpha\mapsto K]}^{\mathcal{T}} \vdash \phi : o}{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash \# \alpha : K.\ \phi : o}\ (\# {\in} \{\forall_\tau,\exists_\tau\})$$

Terms The typing rules for the terms ensure that the declared and used types of variables match their binding quantors and that functions arguments' type match the declaration and are well-typed.

For given term variable mapping $\mathcal{T}$ and type variable mapping $\mathcal{C}$, the following rules determine if a term t is of type τ. Note that all terms of a closed formula share the same mappings $\mathcal{T}$ and $\mathcal{C}$. Furthermore, a (well-typed) function term's type is uniquely determined by its function declaration and its type arguments.

Definition 3.1.2.2 (Term Typing Rules). The typing rules for terms are as follows:

$$\frac{}{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash u : \mathcal{T}(u)}\ (\text{term var})$$

$$\frac{f : \forall \alpha_1{:}K_1, \ldots, \alpha_n{:}K_n.\ \tau_1, \ldots, \tau_m \to \tau \in \mathcal{F} \quad \gamma_{\mathcal{C}}^{\mathcal{T}} \vdash t_1 : \tau_1\sigma \quad \ldots \quad \gamma_{\mathcal{C}}^{\mathcal{T}} \vdash t_m : \tau_m\sigma}{\gamma_{\mathcal{C}}^{\mathcal{T}} \vdash f[\alpha_1\sigma, \ldots, \alpha_n\sigma](t_1, \ldots, t_m) : \tau\sigma}\ (\text{fun})$$

and for all i from 1 to n it holds that $\mathcal{C} \vdash \alpha_i\sigma\ \mathbf{:}\ K_i$

Type Terms The typing rules for type terms infer if a type is a member of a type-class constraint.

For a given type variable mapping $\mathcal{C}$, the following rules determine if a type term τ is a member of a set of type classes K. Note that all type terms of a closed formula share the same mapping $\mathcal{C}$. The $\mathcal{T}$ mentioned in the κ rule, is not a mapping but the type declaration set of the signature.

Definition 3.1.2.3 (Type-Term Typing Rules). The typing rules for type terms are as follows:

$$\frac{}{\mathcal{C} \vdash \tau\ \mathbf{:}\ \emptyset}\ (\text{empty}) \qquad \frac{k \in \mathcal{C}(\alpha)}{\mathcal{C} \vdash \alpha\ \mathbf{:}\ \{k\}}\ (\text{type var})$$

where τ is not a variable

$$\frac{\forall \alpha_1{:}K_1, \ldots, \alpha_n{:}K_n.\ \kappa(\alpha_i, \ldots, \alpha_m) : k \in \mathcal{T} \quad \mathcal{C} \vdash \alpha_1\sigma\ \mathbf{:}\ K_1 \quad \ldots \quad \mathcal{C} \vdash \alpha_n\sigma\ \mathbf{:}\ K_n}{\mathcal{C} \vdash \kappa(\alpha_i\sigma, \ldots, \alpha_m\sigma)\ \mathbf{:}\ \{k\}}\ (\kappa)$$

$$\frac{k_1 \subseteq k_2 \in \mathcal{TC} \quad \mathcal{C} \vdash \tau\ \mathbf{:}\ \{k_1\}}{\mathcal{C} \vdash \tau\ \mathbf{:}\ \{k_2\}}\ (\text{subclass}) \qquad \frac{\mathcal{C} \vdash \tau\ \mathbf{:}\ \{k_1\} \quad \cdots \quad \mathcal{C} \vdash \tau\ \mathbf{:}\ \{k_N\}}{\mathcal{C} \vdash \tau\ \mathbf{:}\ \{k_1, \ldots, k_N\}}\ (\text{combine})$$

where $N > 1$ and τ is not a variable

Non-Overlapping Typing Rules I am now presenting a slight variation on the rules that show that they can be implemented in such a way that always only one typing rule applies. To achieve this, the top typing rule is delayed and the subclass typing rule is integrated in the other type-term typing rules. First, I show that this is sufficient to have non-overlapping typing rules. Obviously, the typing rules of formulas (terms, type terms) can only be applied to formulas (terms, type terms). Thus, it is sufficient to separately show the typing rules for formulas, terms and type terms to be non-overlapping.

Lemma 3.1.2.4. After removing the top rule, all remaining typing rules for formulas (Def. 3.1.2.1) are non-overlapping.

Proof. The conclusion of the formula typing rules can be of one of the following forms:

1. $s \approx t : o_{\mathcal{B}}$,
2. $\phi_1 \# \phi_2 : o_{\mathcal{B}}$ with $\# \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
3. $\neg\phi : o_{\mathcal{B}}$
4. $p[\tau_1, \ldots, \tau_n](t_1, \ldots, t_m) : o_{\mathcal{B}}$.
5. $\#u : \tau,\ \phi : o_{\mathcal{B}}$ with $\# \in \{\forall, \exists\}$
6. $\#u : \tau,\ \phi : o$ with $\# \in \{\forall_\tau, \exists_\tau\}$

Clearly, the structure of these forms is different and they are thus non-overlapping. They also map one-to-one with the typing rules (Def. 3.1.2.1), when not considering the top rule. □

Lemma 3.1.2.5. All typing rules for terms (Def. 3.1.2.2) are non-overlapping.

Proof. The conclusion of the term typing rules can be of one of the following forms:

1. $u : \tau$ or
2. $f[\tau_1, \ldots, \tau_n](t_1, \ldots, t_m) : \tau$.

Again, these forms are non-overlapping and exhaustive and map one-to-one with the typing rules. □

Lemma 3.1.2.6. After removing the subclass typing rule, all remaining typing rules for type terms (Def. 3.1.2.3) are non-overlapping.

Proof. The conclusion of the type-term typing rules can be of one of the following forms:

1. $\tau : \emptyset$,
2. $\alpha : \{k\}$,
3. $\kappa(\alpha_i\sigma, \ldots, \alpha_m\sigma) : \{k\}$ or
4. $\tau : \{k_1, \ldots, k_N\}$ with $N > 1$.

The forms for the type terms are also non-overlapping and exhaustive and map one-to-one with the typing rules, when not considering the subclass rule. Note, in case 4, N is larger than 1, because $N = 1$ is handled by case 2 for type variables and case 3 for type constructors. □

We have shown that, except the top and subclass rules, all typing rules are non-overlapping. Therefore, I now focus on the top and subclass rules.

Top Rule The top rule for formulas only overlaps with the rule for typed quantifiers. It can be delayed until the typed quantifiers have been processed.

Lemma 3.1.2.7. Without loss of generality, the top rule should only be applied if no other rule is applicable.

Proof Sketch. The top rule is only applicable on conclusions of the form $\phi : o$. The only other applicable rule is the one for type quantifiers, which is applicable only if ϕ has a type quantifier at the top. Applying the top rule on a ϕ that has a type quantifier at the top is senseless, because none of the other typing rules is applicable to type quantifiers. Thus, only after exhaustively applying the rule handling type quantifiers, should the top rule be applied.

Subclass Rule To get rid of the overlap between the type var and subclass typing rule and the overlap between the κ and the subclass typing rule they can be combined. The result of the combination of the type var′ and κ typing rules with the subclass typing rule resulting in the following typing rules (and the unchanged empty and combine typing rules)

$$\frac{k' \subseteq^*_{\mathcal{TC}} k \quad k' \in \mathcal{C}(\alpha)}{\mathcal{C} \vdash \alpha : \{k\}} \ \text{(type var}'\text{)}$$

$$\frac{k' \subseteq^*_{\mathcal{TC}} k \quad \forall \alpha_1{:}K_1, \ldots, \alpha_n{:}K_n.\, \kappa(\alpha_i, \ldots, \alpha_m) : k' \in \mathcal{T} \quad \mathcal{C} \vdash \alpha_1\sigma : K_1 \quad \ldots \quad \mathcal{C} \vdash \alpha_n\sigma : K_n}{\mathcal{C} \vdash \kappa(\alpha_i\sigma, \ldots, \alpha_m\sigma) : \{k\}} \ (\kappa')$$

Termination The typing rules terminate, because the recursive premises are always smaller than the conclusion.

Lemma 3.1.2.8. All typing rules assumptions are either non-recursive or assumption on strict subformulas, subterms, subtype-terms or subsets of the type-class constraints of the conclusion.

Determining the type of a term is straight-forward. Either a term is a variable (with annotated type) or a function term, then we can look up its definition and instantiate the definition with the type arguments of the function symbol. We can then check if the term is well-typed by using the above typing rules.

Well-typedness has to be checked only initially, because all operations that are necessary for superposition are well-typedness preserving. From now on, I will only consider well-typed formulas (and clauses) and their terms and type terms.

Notation

After the well-typedness check I annotate all terms with the mappings $\mathcal{T}$ and $\mathcal{C}$ (but never explicitly write them anywhere). After the CNF transformation (Section 3.1.5), only universal quantifier at the top of the formulas remain. Therefore, I can omit all quantifiers and annotate term variables (u) with their type (τ, given by $\mathcal{T}$) and type variables with their type-class constraint (α^K, given by $\mathcal{C}$). I use the same notation (t^τ) to make the type τ of a term t explicit and τ^K to make the type-class constraint (K) of type terms explicit (τ). If the type(-class constraint) is not relevant, I sometimes leave this implicit.

3.1.3. Unification

I now present unification of the polymorphic first-order language with type classes. First I introduce a preprocessing step making unification easier to compute; then I present substitution followed by unification.

Alternative Term Representation The syntax presented above is the input syntax we want to expose to other tools, e.g. Isabelle/HOL. But in that syntax the type of a term $f[\alpha_1\sigma, \ldots, \alpha_n\sigma](t_1, \ldots, t_m)$ is not immediately obvious. In the input syntax, the type of a term must be computed form the type arguments and its declaration $f : \forall\alpha_1{:}K_1, \ldots, \alpha_n{:}K_n.\ \tau_1, \ldots, \tau_m \to \tau \in \mathcal{F}$ (to be $\tau\sigma$). This is unfortunate, because unification is ubiquitous in superposition and for unification we regularly need the type of a term. To simplify unification, I introduce an alternative term representation, that includes the term's type. Additionally, I remove those type arguments that are implicit from the other arguments and thus are not necessary. The alternative representation removes the need to recompute the type of a term each time it is used in unification.

I will first define the alternative representation, then substitutions and then show how to convert the initial term representation to the alternative term representation.

Definition 3.1.3.1 (**Alternative Term Representation**)**.** Let $\mathcal{X}_t$ be a given countably infinite set of term variables. The alternative representation of a term is defined as:

- Every term variable in $\mathcal{X}_t$ is a term
- For all $f \in S_{\mathcal{F}}$ if $t_1, \ldots, t_m$ are terms and $\tau_r, \tau_1, \ldots, \tau_n$ are type terms then $f\langle\tau_r, \tau_1, \ldots, \tau_n\rangle(t_1, \ldots, t_m)$ is also a term.

From here on I will mainly use the alternative term representation. Therefore, *term* will stand only for the alternative term representation, except if explicitly mentioned otherwise.

Substitutions

Here I introduce substitutions for variables, type variables, type terms and terms.

Definition 3.1.3.2 (**Variable Substitutions**)**.** A *substitution* is a mapping from (type) variables to (type) terms.

As before, I use σ, θ for substitutions and write them as $\sigma = \{u_1^{\tau_1} \mapsto t_1, \ldots, \alpha_1^{K_1} \mapsto \tau_1, \ldots\}$, where all u_is and all α_is are pairwise distinct; τ_i must be the same type as the type of t_i and τ_i must be of the type-class constraint K_i. I define *variable substitutions* to be:

$$\sigma(u^\tau) = \begin{cases} t_i & \text{if } u^\tau = u_i^{\tau_i} \\ u & \text{otherwise} \end{cases}$$

A substitution σ is updated to return t for u by

$$\sigma[u \mapsto t](v) = \begin{cases} t & \text{if } v = u \\ \sigma(v) & \text{otherwise} \end{cases}$$

(correspondingly for $\alpha_1 \mapsto \tau_1$ instead of $u_1 \mapsto t_1$). I also write $\alpha\sigma$ for $\sigma(\alpha)$.

Definition 3.1.3.3 (Type Term Substitutions). Substitutions are extended to type terms as follows:

$$\kappa(\tau_1, \ldots, \tau_n)\sigma \ ::= \ \kappa(\tau_1\sigma, \ldots, \tau_n\sigma)$$

Definition 3.1.3.4 (Term Substitutions). Substitutions are extended to terms as follows:

$$\begin{aligned} u^\tau\sigma \ &::= \ \sigma(u^{\tau\sigma}) \\ f\langle\tau_1, \ldots, \tau_n\rangle(t_1, \ldots, t_m)\sigma \ &::= \ f\langle\tau_1\sigma, \ldots, \tau_n\sigma\rangle(t_1\sigma, \ldots, t_m\sigma) \end{aligned}$$

The definitions ensure that the application consists of two steps if σ's domain consists of both type and term variables. Let σ_τ be the part of σ whose domain consists of type variables and σ_t be the remaining part whose domain consists of term variables, then $t\sigma = (t\sigma_\tau)\sigma_t$.

A substitution is *well-typed* if for all its term variable mappings $u_i^{\tau_i} \mapsto t_i$ that τ_i is equal to the type of t_i (i.e. checking with the typing rules that $t_i : \tau_i$) and for all its type variables mappings $\alpha_i^{K_i} \mapsto \tau_i$ that τ_i is a member of K_i, i.e. checking with the typing rules that $\tau_i : K$. I call a substitution σ *grounding* for a given term t if $t\sigma$ contains no variables; a *unifier* of the terms t_1 and t_2 if $t_1\sigma \approx t_2\sigma$; *more general* than σ_2 if for all terms t there exists σ_1 such that $t\sigma\sigma_1 \approx t\sigma_2$; *most general* if for all terms t and forall σ_2 there exists σ_1 such that $t\sigma\sigma_1 \approx t\sigma_2$.

Converting to the Alternative Term Representation To avoid recomputing the type of a term for each unification, I have introduced the alternative term representation. Here I define how to convert the original term representation into the alternative term representation, based only on the information available in the declaration.

Furthermore, for function and predicate terms, some of the type arguments can contain redundant information that can be discarded to obtain a more compact representation. In particular, those type arguments that already occur in a type of an argument of the function can be omitted without any loss of information. Only those type arguments that do not occur in any argument's type must be handled separately. In the original representation of terms, unification requires a lookup of the full type in the declarations in $\mathcal{F}$ (and $\mathcal{P}$) and an instantiation of the declared type, when performing (sub)unification tasks. The alternative representation includes only the information on the necessary type arguments and does not require the lookup of any type in the declarations. Functions and predicates are represented only by their return type, the necessary ‘true’ type argument terms and their argument terms. This negates the need to look up any of the declarations. I only show the transformation for terms but use the same transformation also for predicates. For predicates the return type (τ_r) can be omitted, because it is always the boolean return type for predicates. In detail:

$$f[\tau_1, \ldots, \tau_n](t_1, \ldots, t_m)$$

is represented by

$$f\langle\tau_r, \tau_{i_1}, \ldots, \tau_{i_p}\rangle(t_1, \ldots, t_m)$$

and let $\forall_\tau \alpha_1{:}K_1, \ldots, \alpha_n{:}K_n.\ \tau'_1, \ldots, \tau'_m \to \tau' \in \mathcal{F}$ be f’s declaration. Furthermore, let the substitution σ be such that for all j from 1 to n it holds that $\alpha_j\sigma = \tau_j$ and for all j from 1 to m it holds that the type of t_j is τ'_j. Then τ_r is defined to be $\tau'\sigma$. Let $\alpha_{i_1}, \ldots, \alpha_{i_p}$ be a subsequence of $\alpha_1, \ldots, \alpha_n$ of those α_i that do not occur in $\tau'_1, \ldots, \tau'_m$ or τ'. The $\tau_{i_1}, \ldots, \tau_{i_p}$ is $\alpha_{i_1}\sigma, \ldots, \alpha_{i_p}\sigma$ the instantiation of those type argument variables. Clearly, both representations, together with $\mathcal{F}$ and $\mathcal{P}$, provide the same information and thus can be transformed back and forth. The advantage of the second form is that it requires no lookup and no substitution of declarations in $\mathcal{F}$ (and $\mathcal{P}$) to perform unification between variables and functions. Except for keeping the actual return type around, it is also more compact, because only instantiation of type variables that are not already present in the arguments’ type terms or in the return type term are recorded.

Unification Let t_1, t_2 be two terms or type terms. Then unification of these two terms is the task to derive a substitution σ such that $t_1\sigma = t_2\sigma$. The unification rules work on a tuple $(E; S)$, where E are the equations that are still to be unified and S are the equations that are already processed. I use $\doteq$ for not yet unified equations and $=$ for equations that are already processed. The initial state for unifying t_1 and t_2 is $(\{t_1 \doteq t_2\}; \emptyset)$. If E is empty, the unification is complete and then S contains the unifier, since it only contains variable to (type) term mappings. The unification has failed if any equation in E cannot be removed by applying one of the unification rules. In that case no unification is possible. I define the rules in such a way that there is no overlap between them, this simplifies the proofs below.

The first rules are the rules removing term or type identities:

1 Identity $(t \doteq t, \text{E; S}) \Rightarrow (\text{E; S})$
2 τ-Identity $(\tau \doteq \tau, \text{E; S}) \Rightarrow (\text{E; S})$

The fun- & pred(-Decompose), Orient and Eliminate rules are the typed analogues to the unification rules used in untyped first-order logic. Note that preprocessing has simplified unification for functions and predicates. The remaining rules are only applied if the (τ-)Identity rules are not applicable. First I present the κ-, fun- and pred-Decompose rules:

3 κ $(\kappa(\tau_{1_l}, \ldots, \tau_{n_l}) \doteq \kappa(\tau_{1_r}, \ldots, \tau_{n_r}), \text{E; S}) \Rightarrow$
$(\tau_{l_1} \doteq \tau_{r_1}, \ldots, \tau_{l_n} \doteq \tau_{r_n}, \text{E; S})$
4 fun $(f\langle\tau_{l_1}, \ldots, \tau_{l_n}\rangle(t_{l_1}, \ldots, t_{l_m}) \doteq f\langle\tau_{r_1}, \ldots, \tau_{r_n}\rangle(t_{r_1}, \ldots, t_{r_m}), \text{E; S}) \Rightarrow$
$(\tau_{l_1} \doteq \tau_{r_1}, \ldots, t_{l_1} \doteq t_{r_1}, \ldots, \text{E; S})$
5 pred $(p\langle\tau_{l_1}, \ldots, \tau_{l_n}\rangle(t_{l_1}, \ldots, t_{l_m}) \doteq p\langle\tau_{r_1}, \ldots; \tau_{r_n}\rangle(t_{r_1}, \ldots, t_{r_m}), \text{E; S}) \Rightarrow$
$(\tau_{l_1} \doteq \tau_{r_1}, \ldots, t_{l_1} \doteq t_{r_1}, \ldots, \text{E; S})$

For unification of terms I additionally need the Orient and Eliminate rules. Let *vars*(t) be the set of all (type) variables that are t or subterms of t.

6 Orient	$(t \doteq u, E;\ S)$	$\Rightarrow$	$(u \doteq t, E;\ S)$	t is not a variable
7 Eliminate	$(u^{\tau_u} \doteq t^{\tau_t}, E;\ S)$	$\Rightarrow$	$(\tau_u \doteq \tau_t, E\theta;\ u^{\tau_u} = t^{\tau_t}, S\theta)$	$u \notin vars(t)$ and $\theta = \{u^{\tau_u} \mapsto t^{\tau_t}\}$

For unification of type terms I need the τ-Orient and the Eliminate rules for type variables (α-Eliminate) and non-variable type terms (τ-Eliminate). Those depend on the restricting type terms to fulfill type-class constraints.

A substitution σ is the most general substitution ($mgs_{\tau,K}$) of a type term τ and type-class constraint K if for all σ_2 exists a σ_3 such that if $\tau\sigma_2$ is a member of K (according to $\mathcal{TC}$ and $\mathcal{T}$) then $\tau\sigma_2 = \tau\sigma\sigma_3$. This has no effect on ground type terms and only restricts the type variables' type classes constraint, since restricting a type variable is more general than instantiating it. I.e. a *mgs* is a mapping from type variables to type variables (with potentially stricter type class constraints).

8 τ-Orient	$(\tau \doteq \alpha^K, E;\ S)$	$\Rightarrow$	$(\alpha^K \doteq \tau$,E; S)	τ is not a variable
9 α-Eliminate	$(\alpha^K \doteq \tau, E;\ S)$	$\Rightarrow$	$(E\theta;\ \alpha^K = \tau\sigma, \tau = \tau\sigma, S\theta)$	
				$\tau \in vars(\tau)$, σ is $mgs_{\tau,K}$ and $\theta = \sigma[\alpha^K \mapsto \tau\sigma, \tau \mapsto \tau\sigma]$
10 τ-Eliminate	$(\alpha^K \doteq \tau, E;\ S)$	$\Rightarrow$	$(E\theta;\ \alpha^K = \tau\sigma, S\theta)$	
				$\alpha \notin vars(\tau)$, $\tau \notin vars(\tau)$, σ is $mgs_{\tau,K}$ and $\theta = \sigma[\alpha^K \mapsto \tau\sigma]$

As with untyped first-order unification, an efficient implementation can be archived by using unification contexts instead of substituting E and S. Furthermore, in an actual implementation, rules to detect failures can be easily added to speed up the detection of impossible unifications.

Computing the $mgs_{\tau,K}$ A $mgs_{\tau,K}$ is a substition θ from type variables to type variables (with potentially stricter type class constraints) such that $\tau\theta$ fulfills all type classes of K. Let the union $mgs_{\tau,K} \cup mgs'_{\tau',K'}$ of two such substitutions $mgs_{\tau,K} = \{\alpha_1^{K_1} \mapsto \alpha_1^{K_1'}, \dots\}$ and $mgs_{\tau',K'} = \{\alpha_1^{K_1} \mapsto \alpha_1^{K_1''}, \dots\}$ be $\{\alpha_1^{K_1} \mapsto \alpha_1^{K_1' \cup K_1''}, \dots\}$.

The $mgs_{\tau,K}$ (if it exists) can be recursively computed by the following case distinction on τ:

$\tau = \alpha^{K_\alpha}$	then $mgs_{\tau,K}$ maps $\alpha^{K_\alpha} \mapsto \alpha^{K \cup K_\alpha}$ if $K \cup K_\alpha$ is populated (i.e. contains a ground type).
$\tau = \kappa(\tau_1, \dots, \tau_n)$	For each $k \in K$, there is either a type declaration for κ and k with the most general type constraints (compared to all subclasses) or there is a subclass κ and k' that has the most general type constraints (compared to all subclasses). This is ensured by coregularity (Def. 3.1.1.8). Let $\forall \dots \alpha_j : K_j \dots \kappa(\dots, \alpha_j$ at argument position i,$\dots) : k \in \mathcal{T}$ be the corresponding type declaration, then $mgs_{\tau,\{k\}}$ is recursively defined by $mgs_{\tau_1,K_1} \cup \dots \cup mgs_{\tau_n,K_n}$, where τ_i are the arguments of κ and K_i are the K_i corresponding to the type variable at the ith position in the type declaration. Then $mgs_{\tau,K}$ is $\bigcup_{k \in K} mgs_{\tau,\{k\}}$.

Correctness and Uniqueness of Unification I now show that the presented unification computes unifiers (Theorem 3.1.3.11) and that the most general unifier is unique (Theorem 3.1.3.12).

First we have to show that S, the second part of the unification tuple, always contains a substitution (Lemma 3.1.3.6). This requires us to show that S's variables do not occur in E (Lemma 3.1.3.5). Then we show that the $mgs_{\tau,K}$ is most general and unique (Lemma 3.1.3.7). Further properties that we require are that unification terminates (Lemma 3.1.3.8), that at each point in the unification process exactly one unification rule is applicable to an given equation (Lemma 3.1.3.9) and crucially that each unification step preserves the unifiers (Lemma 3.1.3.10). From those lemmas, I can finally conclude that the polymorphic first-order language with type classes has a most general unifier which is unique up to renaming of the (type) variables.

I start by showing that the variables occurring in S do not occur in E, the first part of the unification tuple.

Lemma 3.1.3.5 (S's variables not in E)**.**
Let $(E;\ S) \Rightarrow (E';\ S')$ be one unification step. If E does not contain any variable that occurs in the left-hand side of an equation of S, then E' does not contain any variable that occurs in the left-hand side of S'.

Proof. By induction over the unification rules.

The (τ-)Identity, κ-, fun- and pred-Decompose and (α-)Orient rules do not change S at all, i.e. $S = S'$. The (τ-)Identity rules removes an equation from E to derive E' and thus cannot add additional variables to E' that were not present in E. The κ-, fun- and pred-Decompose remove an equation, with the same top level symbol on both sides, from E and add equations between the arguments to E'. Since the variables of the arguments are also contained in the removed equation, no new variables are introduced into E'. The (α-)Orient rules remove an equation from E and add it with the two sides of the equations swapped to E', clearly no new variables can be introduced this way.

The Eliminate rule introduces a new left-hand side variable u in S' but instantiates all of u's occurrences in E'. It also instantiates u in the remainder of S', but by induction hypothesis u does not occur on the left hand side of the substitution represented by S.

The α-Eliminate rule introduces two new left-hand side type variables α and τ in S' but instantiates all of their occurrences in E' to a type fresh variable. Note, that the substitution σ possibly instantiates variables occurring in τ to make the type-class constraint K of α and the type-class constraint of τ match. The α-Eliminate rule also instantiates the α, τ and the type variables from the substitution σ in the remainder of S' but by induction hypothesis they do not occur on the left hand side of the substitution represented by S.

The τ-Eliminate is analog to the α-Eliminate, except that τ does not become a left-hand side in S'. □

To be a unifier, the computed mapping S must be a substitution, because a unifier is a substitution that makes two terms syntactically equal.

Lemma 3.1.3.6 (S contains a substitution)**.**
Let t_1, t_2 be two terms and $(t_1 \doteq t_2;\ \emptyset) \Rightarrow^* (E;\ S)$ be a series of unification steps then S only contains variable to (type) term mappings.

Proof. By induction over the unification rules.

The (τ-)Identity, κ-, fun- and pred-Decompose and (α-)Orient rule do not change S at all.

The Eliminate and α- and τ-Eliminate rules introduce a new equations in S. The left hand side of those equations are variables. Those variables can never be instantiated because after the application of the rule they do not occur in E anymore (Lemma 3.1.3.5). □

For the unification algorithm to work, the unification of the type-class (constraints) has to be most general and unique, up to renaming of the variables.

Lemma 3.1.3.7. $mgs_{\tau,K}$ is most general and unique (up to renaming).

Proof. Proof by induction over τ.

- Let τ be α^{K_α} then $mgs_{\tau,K}$ is of the most general type-class constraint K_u unifying K_α and K, i.e. $K_u = K_\alpha \cup K$. Either K_u is not populated, meaning that there is no type constructor κ which has declarations (in $\mathcal{T}$ and $\mathcal{TC}$) for all $k \in K_u$ or the $mgs_{\tau,K}$ is $\alpha' : K_u$, which is unique and most general (up to renaming).

- Let τ be $\kappa(\tau_1, \ldots, \tau_n)$. For each type class $k \in K$ coregularity requires that there exists a type declaration $\forall \ldots \alpha_j : K_j \ldots \kappa(\ldots, \alpha_j$ at argument position i$, \ldots) : k' \in \mathcal{T}$ that is more general than the type declaration of all subclasses (k' is k if there is a type declaration for κ and k). The recursive requirements mgs_{τ_i,K_j} are either empty or unique and most general by the induction hypothesis and so is the union of the requirements.

□

The unification algorithm must terminate and at each step only one rule may be applicable.

Lemma 3.1.3.8 (Unification Terminates)**.**
Applying unification on $(\{t_1 \doteq t_2\};\ \emptyset)$ always terminates.

Proof. Applying unification decreases the lexicographic ordering on the pair of number of distinct variables and number of symbols occurring in E below a $\doteq$ obligation. This is not true for the Orient rule, but it can only be applied at most once for each $\doteq$ obligation. □

Knowing that only one unification rule is applicable per equation simplifies the following proofs and thus we show that property now.

Lemma 3.1.3.9 (At any point in the unification only one unification rule is applicable per equation)**.**
If $(t \doteq s, E';\ S)$ and t unifiable with s, then exactly one rule of the unification algorithm is applicable on $t \doteq s$.

Proof. Unification is only possible between predicates, between terms or between types. There is only the pred-Decompose rule for predicates, thus only that rule can apply.

Suppose that both t and s are terms. If $t = s$ then, per definition, only the Identity rule is applicable. Otherwise, if $t \neq s$ then there are four cases: either t and s have a function symbol at the top, one of them is a variable and the other is not, or both of them are a variable:

- Let $t = f\langle\tau_1, \ldots, \tau_n\rangle(t_1, \ldots, t_m)$ and $s = g\langle\tau_{s_1}, \ldots, \tau_{s_n}\rangle(s_1, \ldots, s_m)$ both with function symbols at the top. Then they are only unifiable if the top function symbols ($f = g$) and the type terms and terms are unifiable. In this case, only the fun-Decompose rule is applicable.

- Let $t = f\langle\tau_1, \ldots, \tau_n\rangle(t_1, \ldots, t_m)$ have a function symbol at the top and $s = u$ be a variable. In this case, only the Orient rule is applicable.

- Let $t = u$ be a variable and $s = g\langle\tau_1, \ldots, \tau_n\rangle(s_1, \ldots, s_m)$ have a function symbol at the top. Then they are unifiable only if s does not contain t. In this case, only the Eliminate rule is applicable.

- Let $t = u_t$ and $s = u_s$ both be variables. In this case, only the Eliminate rule is applicable.

Suppose that both t and s are type terms. If $t = s$ then only the τ-Identity rule is applicable. Otherwise, if $t \neq s$ then there are four cases either t and s have a type symbol at the top, one of them is a variable and the other is not, or both of them are a variable:

- Let $t = \kappa_t(\tau_{t_1}, \ldots, \tau_{t_n})$ and $s = \kappa_s(\tau_{s_1}, \ldots, \tau_{s_n})$ both have type symbols at the top. Then they are only unifiable if the top type symbols ($\kappa_t = \kappa_s$) are unifiable. In this case, only the κ-Decompose rule is applicable.

- Let $t = \kappa_l(\tau_1, \ldots, \tau_n)$ have a type symbol at the top and $s = \alpha$ be a type variable. In this case, only the τ-Orient rule is applicable.

- Let $t = \alpha$ be a variable and $s = \kappa_r(\tau_1, \ldots, \tau_n)$ have a type symbol at the top. Then they are unifiable only if s does not contain t. In this case, only the τ-Eliminate rule is applicable.

- Let $t = \alpha_t$ and $\alpha = u_s$ both be type variables. In this case only the α-Eliminate rule can be applicable.

□

To show that the unification algorithm computes the most general unifier, I show that if for each step of the unification algorithm no unifier is lost, then the most general unifier must be an instance (renaming) of the resulting substitution.

Lemma 3.1.3.10 (Unification Preserves Unifiers)**.**
Let $(E; S) \Rightarrow (E'; S')$ be one step of the unification algorithm. Then if θ is a unifier for $(E; S)$ then there is also a unifier θ_2 of $(E'; S')$ such that for the substitution σ contained in S' it holds that $\theta = \sigma\theta_2$.

Proof. Let $t \doteq s$ be an arbitrary unification obligation in E. Let E_r be E without $t \doteq s$. Then the proof is by induction over the rules of the unification algorithm

- (τ-)Identity, then $t = s$ and $E' = E_r$ and $S' = S$. Clearly, if θ is a unifier for $(E; S)$ then it is also a unifier for $(E'; S')$.

- κ-Decompose, then $t = \kappa(\tau_1, \ldots, \tau_n)$ and $s = \kappa(\tau_{s_1}, \ldots, \tau_{s_n})$ and E' is $\tau_1 \doteq \tau_{s_1}, \ldots, \tau_m \doteq \tau_{s_m}, E_r$ and $S' = S$. Clearly, if θ is a unifier for $(E; S)$ then $t\theta = s\theta$ and thus it is also a unifier for the subtypes $\tau_i\theta = \tau_{s_i}\theta$. Therefore, θ is also a unifier for $(E'; S')$.

- fun-Decompose, then $t = f\langle\tau_1, \ldots, \tau_m\rangle(t_1,\ldots,t_m)$ and $s = f\langle\tau_{s_1}, \ldots, \tau_{s_m}\rangle(s_1, \ldots, s_m)$ and E' is $\tau_1 \doteq \tau_{s_1}, \ldots, \tau_m \doteq \tau_{s_m}, t_1 \doteq s_1, \ldots, t_m \doteq s_m, E_r$ and $S' = S$. Clearly, if θ is a unifier for $(E;\ S)$ then $t\theta = s\theta$ and thus it is also a unifier for the types $\tau_i\theta = \tau_{s_i}\theta$ and for the subterms $t_i\theta = s_i\theta$. Therefore, θ is also a unifier for $(E';\ S')$.

- pred-Decompose, then $t = p\langle\tau_1, \ldots, \tau_m\rangle(t_1, \ldots, t_m)$ and $s = p\langle\tau_{s_1}, \ldots, \tau_{s_m}\rangle(s_1, \ldots, s_m)$ and E' is $\tau_1 \doteq \tau_{s_1}, \ldots, \tau_m \doteq \tau_{s_m}, t_1 \doteq s_1, \ldots, t_m \doteq s_m, E_r$ and $S' = S$. Clearly, if θ is a unifier for $(E;\ S)$ then $t\theta = s\theta$ and thus it is also a unifier for the types $\tau_i\theta = \tau_{s_i}\theta$ and for the subterms $t_i\theta = s_i\theta$. It is therefore also a unifier for $(E';\ S')$.

- Orient rule, then $t = f\langle\tau_1, \ldots, \tau_m\rangle(t_1, \ldots, t_m)$, $s = u$, E' is $s \doteq t, E_r$ and $S' = S$. Clearly, if θ is a unifier of $(t \doteq s, E_r;\ S)$ it is also a unifier of $(t \doteq s, E_r;\ S)$

- Eliminate rule, then t is a variable (u) and of type τ_t, u is not a subterm of s, s is of type τ_s, E' is $\tau_t \doteq \tau_s$, $E_r\{u \mapsto s\}$ and S' is $S\{u \mapsto s\}$. Clearly, if θ is a unifier of $(E;\ S)$ then $u\theta = s\theta$ and thus their types are equal ($\tau_t\theta = \tau_s\theta$). Furthermore, θ can be split into $\theta = \theta_1\theta_2$ such that $u\theta_1 = s$, i.e. $\theta_1 = \{u \mapsto s\}$, which is the substitution used to derive E' and S'. Then θ_2 is a unifier of $(E';\ S')$.

- τ-Orient rule, then $t = \kappa(\tau_1, \ldots, \tau_m)$, $s = \alpha$, E' is $s \doteq t, E_r$ and $S' = S$. Clearly, if θ is a unifier of $(t \doteq s, E_r;\ S)$ it is also a unifier of $(t \doteq s, E_r;\ S)$

- α-Eliminate rule, then $t = \alpha_t$, $s = \alpha_s$, σ is the *mgs* of s and t, E' is $E_r\sigma$ and S is $S'\sigma$. If θ is a unifier for $(E;\ S)$ then $t\theta = s\theta$ and t and s are (at least) instantiated according to their *mgs* (possibly further), therefore, θ can be split into $\theta = \theta_1\theta_2$ such that θ_1 is the *mgs* of s and t, which is the substitution used to derive E' and S'. Clearly, θ_2 is a unifier of $(E';\ S')$.

- τ-Eliminate rule, then $t = \alpha$, $s = \kappa(\tau_1, \ldots, \tau_m)$, σ is the *mgs* of s and t, E' is $E_r\sigma$ and S is $S'\sigma$. If θ is a unifier for $(E;\ S)$, then $t\theta = s\theta$ and t and s are (at least) instantiated according to their *mgs* (possibly further), therefore, θ can be split into $\theta = \theta_1\theta_2$ such that θ_1 is the *mgs* of s and t, which is the substitution used to derive E' and S'. Clearly, θ_2 is a unifier of $(E';\ S')$.

□

Now I can combine the previous lemmas to show that a most general unifier is computed and that it is unique up to renaming of the variables.

Theorem 3.1.3.11 (Unification Preserves Unifier)**.**
If there is a unifier θ of t and s then $(\{t \doteq s\};\ \emptyset) \Rightarrow^* (\emptyset,\ S)$ and θ is an instance of the unifier contained in S.

Proof. Suppose there is a unifier and unification does not result in a $(\emptyset,\ S)$ that contains that unifier. Since the unification algorithm always terminates (Lemma 3.1.3.8) there are the following possibilities:

1. $(\{t \doteq s\};\ \emptyset) \Rightarrow^* (E,\ S)$ and $E \neq \emptyset$, or

2. $(\{t \doteq s\};\ \emptyset) \Rightarrow^* (\emptyset,\ S)$ and S does not contain the most general unifier.

In the first case, Lemma 3.1.3.9 guarantees that there is a unification rule that is applicable if E is unifiable. E must be unifiable if there is a most general unifier, but then $(E,\ S)$ cannot be the end result of the unification algorithm. In the second case, no unifier (in particular the most general unifier) is lost by a step of the unification algorithm (Lemma 3.1.3.10). Lemma 3.1.3.6 guarantees that S always contains a substitution. Therefore, the unifier is an instance of the unifier contained in S. □

Theorem 3.1.3.12 (The Most General Unifier is Unique)**.**
The most general unifier of two terms t and s is unique up to renaming of variables.

Proof. Follows from Lemma 3.1.3.11, because all most general unifiers are instances of the unifier θ contained in S. □

3.1.4. Semantics

So far, I have presented the syntax, the typing rules and the unification algorithm for the polymorphic first-order language. Here I give the semantics. In contrast to the semantic for untyped first-order language, the universe now has to be partitioned into domains. Domains are disjoint subsets of the universe. Those domains then represent the interpretation of the types. Terms are interpreted as elements of the universe. The interpretation of a term must be an element of the domain that corresponds to the type of the term. Type classes and type-class constraints are interpreted as a subset of the set of domains for which their type-class declarations are fulfilled. The domains which are fulfilled by a type class are defined by a fixpoint over the type-class declarations in $\mathcal{T}$ and $\mathcal{TC}$.

Definition 3.1.4.1 (**Structure**)**.** Given a signature Σ for polymorphic first-order language with type classes, the corresponding Σ-structure is a tuple $\Sigma = (\mathcal{U},\ \mathcal{D},\ \mathcal{I}_\mathcal{T},\ \mathcal{I}_\mathcal{F},\ \mathcal{I}_\mathcal{P})$ where

$\mathcal{U}$ is a non-empty countable set of elements, the universe.
$\mathcal{D}$ a non-empty set of non-empty disjoint subsets of $\mathcal{U}$. It represents the set of the types (domains).
$\mathcal{I}_\mathcal{T}$ is the set of type constructors ($\kappa^\mathcal{I}$) which map domains to a domain ($\mathcal{D}^n \to \mathcal{D}$).
$\mathcal{I}_\mathcal{F}$ is the set of functions ($f^\mathcal{I}$) which maps a cartesian product of domains and elements to an element ($\mathcal{D}^n \times \mathcal{U}^m \to \mathcal{U}$).
$\mathcal{I}_\mathcal{P}$ is the set of predicates ($p^\mathcal{I}$) which map a cartesian product of domains and elements to true or false ($\mathcal{D}^n \times \mathcal{U}^m \to \{0,\ 1\}$).

Note that the set $\mathcal{D}$ is countable, because there are only countable many non-empty disjoint subsets of a countable set (there are only countable many disjoint subsets of size one) and the universe $\mathcal{U}^T$ is countable.

Interpretation A variable valuation is a mapping from variables to elements of the universe relative to a given Σ-structure. In general, I will not explicitly mention the Σ-structure but leave this relation implicit. I use θ to map type variables to domains ($\mathcal{X}_\tau \to \mathcal{D}$) and ξ to map term variables to elements ($\mathcal{X}_t \to \mathcal{U}$).

Interpretations ($\mathcal{I}_{\theta,\xi}$) of type terms and terms are in general parameterized by both the type variable (θ) and the term variable (ξ) mapping. If I omit one or both mappings, they are assumed to be the empty mapping. For example, the interpretation of type terms only requires the mapping for type variables (θ) and therefore omit the term variable mapping.

Definition 3.1.4.2 (Interpretation of Type Terms). The interpretation of a type term is as follows:

$$\begin{aligned} \mathcal{I}_\theta(\alpha) &= \theta(\alpha) \\ \mathcal{I}_\theta(\kappa(\tau_1, \ldots, \tau_n)) &= \kappa^{\mathcal{I}}(\mathcal{I}_\theta(\tau_1), \ldots, \mathcal{I}_\theta(\tau_n)) \end{aligned}$$

where $\kappa^{\mathcal{I}} \in \mathcal{I}_\mathcal{T}$ is the type constructor corresponding to the type symbol $\kappa \in S_\mathcal{T}$. Additionally, each ground term must evaluate to a different $d \in \mathcal{D}$.

I also use the symbol $\mathcal{I}$ for valuations of terms. Term valuations are parameterized by two valuations, one for type variables (θ) and one for term variable (ξ).

Definition 3.1.4.3 (Interpretation of Terms). The interpretation of a term is as follows:

$$\begin{aligned} \mathcal{I}_{\theta,\xi}(u) &= \xi(u) \\ \mathcal{I}_{\theta,\xi}(f\langle\tau_i, \ldots, \tau_n\rangle(t_1, \ldots, t_m)) &= f^{\mathcal{I}}\langle\mathcal{I}_\theta(\tau_1), \ldots, \mathcal{I}_\theta(\tau_n)\rangle(\mathcal{I}_{\theta,\xi}(t_1), \ldots, \mathcal{I}_{\theta,\xi}(t_m))) \end{aligned}$$

where $f^{\mathcal{I}} \in \mathcal{I}_\mathcal{F}$ is the function corresponding to the function symbol $f \in S_\mathcal{F}$. Additionally, $f^{\mathcal{I}}\langle\mathcal{I}_\theta(\tau_1), \ldots, \mathcal{I}_\theta(\tau_n)\rangle(\mathcal{I}_{\theta,\xi}(t_1), \ldots, \mathcal{I}_{\theta,\xi}(t_m))$ must evaluate to an element of $\mathcal{I}_\theta(\tau_r)$, where τ_r is the type assigned to $f\langle\tau_i, \ldots, \tau_n\rangle(t_1, \ldots, t_m)$ by the typing rules (Def. 3.1.2.2).

The interpretation for type-class constraints (K) need no parameters and also results in a set of domains that type-class constraint represents.

Definition 3.1.4.4 (Interpretation of Type-Class Constraints). The interpretation of a type-class constraint is as follows:

1. $\mathcal{I}(K) = \mathcal{D}$ if $K = \emptyset$
2. $\mathcal{I}(K) = \bigcap_{k \in K} \mathcal{I}(k)$ if $K \neq \emptyset$

Note that $\mathcal{I}(K)$ may be empty. This is the case when there is no ground type that is an element of all type-classes in K. The interpretation of a type class is a fixpoint defined by the interpretations of the types and the type class declarations. The fixpoint results in a set of domains that the type class represents.

Definition 3.1.4.5 (Interpretation of Type Classes). The set of domains that form the interpretation of a type class k is defined by the least fixpoint of the type-class constraint interpretation and the following two rules, i.e. $\mathcal{I}(k)$ is defined as the least fixpoint of:

1. For all declarations $\forall_\tau\ \alpha_1{:}K_1, \ldots, \alpha_n{:}K_n.\ \tau{:}k \in \mathcal{T}$ and all grounding substitution σ of τ it holds that
$$\left(\bigwedge_{1 \le i \le n} \mathcal{I}(\alpha_i\sigma) \in \mathcal{I}(K_i) \right) \Rightarrow \mathcal{I}(\tau\sigma) \in \mathcal{I}(k)$$

2. For all declarations $k_1 \subseteq k \in \mathcal{TC}$ and all ground type terms τ, I require that if a type is an element of a type class, then it is also element of its superclass:

$$\mathcal{I}(\tau) \in \mathcal{I}(k_1) \Rightarrow \mathcal{I}(\tau) \in \mathcal{I}(k)$$

Finally, interpretations of formulas, like interpretations of terms, need two parameters, one for type variables (θ) and one for term variable (ξ). They are adapted by the quantifications occurring in the formulas. Let false be represented by 0 and true be represented by 1 and let *max*, *min* the corresponding functions for maximum and minimum. The interpretation of formulas is defined by

Definition 3.1.4.6 (Interpretation of Formulas). The interpretation of a formula is as follows:

$$\begin{array}{lcl}
\mathcal{I}_{\theta,\xi}(p\langle\tau_{\alpha_i}, \ldots, \alpha_n\rangle(t_1, \ldots, t_m)) & = & p^{\mathcal{I}}[\mathcal{I}_\theta(\tau_{\alpha_1}), \ldots, \mathcal{I}_\theta(\tau_{\alpha_n})](\mathcal{I}_{\theta,\xi}(t_1), \ldots, \mathcal{I}_{\theta,\xi}(t_m)) \\
\mathcal{I}_{\theta,\xi}(\bot) & = & 0 \\
\mathcal{I}_{\theta,\xi}(\top) & = & 1 \\
\mathcal{I}_{\theta,\xi}(s \approx t) & = & 1 \Leftrightarrow \mathcal{I}_{\theta,\xi}(s) = \mathcal{I}_{\theta,\xi}(t) \\
\mathcal{I}_{\theta,\xi}(\neg\phi) & = & 1 - \mathcal{I}_{\theta,\xi}(\phi) \\
\mathcal{I}_{\theta,\xi}(\phi \wedge \psi) & = & \min(\mathcal{I}_{\theta,\xi}(\phi), \mathcal{I}_{\theta,\xi}(\psi)) \\
\mathcal{I}_{\theta,\xi}(\phi \vee \psi) & = & \max(\mathcal{I}_{\theta,\xi}(\phi), \mathcal{I}_{\theta,\xi}(\psi)) \\
\mathcal{I}_{\theta,\xi}(\phi \rightarrow \psi) & = & \max(1 - \mathcal{I}_{\theta,\xi}(\phi), \mathcal{I}_{\theta,\xi}(\psi)) \\
\mathcal{I}_{\theta,\xi}(\phi \leftrightarrow \psi) & = & 1 \Leftrightarrow \mathcal{I}_{\theta,\xi}(\phi) = \mathcal{I}_{\theta,\xi}(\psi) \\
\mathcal{I}_{\theta,\xi}(\forall u : \tau.\ \phi) & = & \min\limits_{e \in \mathcal{I}_\theta(\tau)} (\mathcal{I}_{\theta,\xi[u \to e]}(\phi)) \\
\mathcal{I}_{\theta,\xi}(\exists u : \tau.\ \phi) & = & \max\limits_{e \in \mathcal{I}_\theta(\tau)} (\mathcal{I}_{\theta,\xi[u \to e]}(\phi)) \\
\mathcal{I}_{\theta,\xi}(\forall_\tau\ \alpha : K.\ \phi) & = & \min\limits_{d \in \mathcal{I}(K)} (\mathcal{I}_{\theta[\alpha \to d],\xi}(\phi)) \\
\mathcal{I}_{\theta,\xi}(\exists_\tau\ \alpha : K.\ \phi) & = & \max\limits_{d \in \mathcal{I}(K)} (\mathcal{I}_{\theta[\alpha \to d],\xi}(\phi))
\end{array}$$

where $p^{\mathcal{I}} \in \mathcal{I}_{\mathcal{P}}$ is the function corresponding to the function symbol $p \in S_{\mathcal{P}}$.

Properties of the Semantics

To show that the rules of definition 3.1.4.5 have a least fixpoint, I use Tarski's fixpoint theorem. Therefore, I first present his definition of complete lattice, increasing function and then Tarski's fixpoint theorem [62]:

Definition 3.1.4.7 (Complete Lattice [62]). A *complete lattice* is a system $\langle A, \leq\rangle$ such that A is a non-empty set and $\leq$ a partial order on elements of A such that for each $B \subseteq A$ there is a least upper bound of B and a least lower bound of B.

Definition 3.1.4.8 (Increasing Function [62]). A function f, $A \mapsto A$, is *increasing* if for any two elements $a, b \in A$ it holds that $a \leq b$ implies $f(a) \leq f(b)$.

Theorem 3.1.4.9 (Tarski's Lattice-Theoretical Fixpoint Theorem [62]).
Let $\langle A, \leq\rangle$ be a complete lattice, f be an increasing function $A \mapsto A$ and P be the set of all

fixpoints of f. Then the set P is not empty and the system $\langle P, \leq\rangle$ is a complete lattice, in particular f has a least fixpoint and a greatest fixpoint.

I now use Tarski's fixpoint theorem to show the existence of a least fixpoint defining $\mathcal{I}(k)$.

Lemma 3.1.4.10 (The Least Fixpoint Exists)**.**
The defining rules of $\mathcal{I}(k)$ have a least fixpoint.

Proof. Let the set A be the set of all subsets of $\mathcal{D}$. Let $\leq$ be a partial order, such that for a, $b \in A$ it holds that $a \leq b$ if there are more domains in b than in a. The upper bound of a subset B of A is the union of the sets in B and the lower bound is the intersection of the sets in B. Then clearly $\langle A, \leq\rangle$ is a complete lattice. The function f, $A \mapsto A$, is (implicitly) given by the fixpoint rules, by mapping the current interpretation ($\mathcal{I}(k)$) to a new extended set of domains as long as one of the rules is applicable. Thus, we have to show both rules (Def. 3.1.4.5) increasing:

1. Rule one adds the type definitions to their corresponding type classes, thereby increasing the number of elements.

2. Rule two adds the subclass definitions to the corresponding superclasses, thereby increasing the number of elements.

Since the interpretation of the type classes also includes the interpretation rules of the type-class constraints (Def. 3.1.4.4), we have to show both of those rules increasing:

1. Rule one does not change at all, thus it fulfills the requirements of increasing the number of elements.

2. Rule either stays the same or adds new elements if a $\mathcal{I}(k)$ grows such that the whole intersection contains a new domain, thus it is also increasing.

Thus, by Theorem 3.1.4.9 the least fixpoint exists. □

Now I show that the judgment of the typing rules matches the semantic I have just defined.

Lemma 3.1.4.11 (Typing Matches Type Constraint Semantics)**.**
For each type term τ and type-class constraint K, if the judgment $\tau : K$ holds, according to the typing rules, then $\mathcal{I}(\tau) \in \mathcal{I}(K)$.

Proof. Proof by induction over the typing rules for ground type terms.

1. empty, i.e. $\tau : \emptyset$. All type terms are of the empty type-class constraint, and its interpretation is $\mathcal{D}$.

2. type var, i.e. $\alpha : \{k\}$. Then by the definition of the type var typing rule α has the type-class constraint K such that $k \in K$. The interpretation of α is $\mathcal{I}(\alpha) = \theta(\alpha)$ and $\theta(\alpha)$ is bound by the semantic of the type quantification of α to a domain $d \in \mathcal{I}(K)$.

3. κ, i.e. $\tau : \{k\}$ and there is a declaration $(\forall \alpha_1{:}K_1, \ldots, \alpha_n{:}K_n.\, \kappa(\alpha_i, \ldots, \alpha_m) : k \in \mathcal{T})$ and from the induction hypothesis we know that for the premises the lemma holds (i.e. $\mathcal{I}(\alpha_i\sigma : K_i) \in \mathcal{I}(K_i)$). Then by rule one of the type class interpretation, the interpretation of τ is part of the interpretation of k, i.e. $\mathcal{I}(\tau) \in \mathcal{I}(K)$.

4. subclass, i.e. $\tau : \{k\}$ and $k_1 \subseteq k$ and by the second rule of the type class interpretation, the interpretation of τ is contained in the interpretation of k.

5. combine, i.e. $\tau : K$ and K has at least two elements. From the induction hypothesis we know that for each $k_i \in K$ that $\tau : \{k_i\}$. Then for each $k_i \in K$ that $\mathcal{I}(\tau) \in \mathcal{I}(k_i)$ and thus $\mathcal{I}(\tau) \in \bigcap_{k_i \in K} \mathcal{I}(k_i)$ and thus $\mathcal{I}(\tau) \in \mathcal{I}(K)$.

□

Furthermore, I show that for each interpretation of a type class k if there is a domain d in that type class k then there is a type term τ such that the interpretation of τ is d.

Lemma 3.1.4.12 (Type Class Are Type Term Generated)**.**
Let k be a type class, if $d \in \mathcal{I}(k)$ then there is a ground type term τ such that $\tau : \{k\}$ holds, according to the typing rules and $\mathcal{I}(\tau) = d$.

Proof. If $d \in \mathcal{I}(k)$ then it was introduced by one of the (least) fixpoint rules of the type class semantics. Therefore, the proof is by induction on the fixpoint rules of the type class semantics.

1. If d was introduced by the first rule then there is a ground type term $\tau\sigma$ such that $\mathcal{I}(\tau\sigma) = d$.

2. If d was introduced by the second rule then there is a ground type term τ such that $\mathcal{I}(\tau) = d$.

□

From this it then follows that the semantic of type-class constraints matches the judgment of the typing rules.

Lemma 3.1.4.13 (Type-Class Constraint Semantics Matches Typing)**.**
Let $K \neq \emptyset$ be a type-class constraint, if $d \in \mathcal{I}(K)$ then there is a ground term τ such that $\tau : K$ holds, according to the typing rules and $\mathcal{I}(\tau) = d$.

Proof. By definition of the semantics of type-class constraints $\mathcal{I}(K) = \bigcap_{k \in K} \mathcal{I}(k)$. For each $k \in K$ and $d \in \mathcal{I}(k)$ there exists a ground type term $\tau : \{k\}$ (Lemma 3.1.4.12). Since, by the definition of type term interpretations, different ground type terms evaluate to a different (disjoint) domains, we have one ground type term τ for each domain $d \in \mathcal{I}(K)$ such that for all $k \in K$ that $\tau : \{k\}$. Thus from the combine typing rule it follows that $\tau : K$. □

Finally, I can prove that the interpretation of a type-class constraint is the union of the interpretations of the type-class constraint's ground type terms. We require this property later, in the refutational completeness proof, in particular in the lifting step from the monomorphic to the polymorphic first-order logic with type classes.

Corollary 3.1.4.14 (Type-Class Constraint Interpretations are the Union of Ground Type Term Interpretations)**.**
The interpretation of a type-class constraint $K \neq \emptyset$ is the union of the interpretations of its ground type terms: $\mathcal{I}(K) = \{\mathcal{I}(\tau) \mid \tau : K \text{ and } \tau \text{ is ground}\}$.

Proof. Follows from Lemmas 3.1.4.13 and 3.1.4.11. □

3.1.5. Clausification and Skolemization

The semantics of the boolean connectives for typed formulas is not different from the semantics in the untyped case. Thus, the prenex normal form and clause normal form translations are performed as before (Section 2.1.3), with the only difference that there are two separate sets of quantifiers - the type and term quantifier. Skolemization, however, must be adapted to the typed setting.

Type Skolemization

In principle, before clausification, all type quantifications with empty type-class constraint would need to be removed ($\Rightarrow_{E_\tau}$). This is because I restricted the search space to only consider combinations of type classes that do contain a ground type.

$\forall_\tau \alpha{:}K.\ \phi \Rightarrow_{E_\tau} \top$ if there is no ground type that fulfills K

$\exists_\tau \alpha{:}K.\ \phi \Rightarrow_{E_\tau} \bot$ if there is no ground type that fulfills K

However, in our target use case, the translation of proof obligations from a higher-order proof assistant to a first-order automated theorem prover, it is impractical to translate all facts and axioms known to the proof assistant. Consider, for example, the case where we want to show a property holds for all members of a type-class constraint (e.g. the negated conjecture $\exists_\tau \alpha{:}K.\neg\phi$) and the perfect matching axiom (e.g. $\forall_\tau \alpha{:}K.\phi$). But, because no ground type was included both the conjecture and the axiom would be removed. In fact, including a ground type would only need to be done to populate the type classes, but could not possibly be helpful in the actual reasoning process. Adding all necessary ground types would require the proof assistant to analyze each proof obligation and accompanying facts, without any reasonable gain. To avoid this, I assume that all type-class constraints in the initial axioms and conjecture are supposed to be populated, i.e. I keep the formulas even though Skolemization may then produce Skolem type constructors for previously unpopulated type classes. This removes the need for the proof assistant to spend effort to specify ground types. While this makes Skolemization not satisfiability preserving (by potentially introducing new ground types in type classes with no ground types), I feel this is a reasonable compromise. Especially, since it is anyways not reasonable for the proof assistant to pass the complete type hierarchy and all axioms to the automated theorem prover.

Definition 3.1.5.1 (Type Skolemization). The Skolem transformation for types ($\Rightarrow_{S_\tau}$) is:

$$\forall_\tau \alpha_1{:}K_1,\ \ldots,\ \forall_\tau \alpha_n{:}K_n.\ \exists_\tau \alpha{:}K_\alpha.\ \phi$$
$$\Rightarrow_{S_\tau}$$
$$\forall_\tau \alpha_1{:}K_1,\ \ldots,\ \forall_\tau \alpha_n{:}K_n.\ \phi\{\alpha \mapsto \kappa(\alpha_1,\ \ldots,\ \alpha_n)\}$$

where $n \geq 0$ and κ is a type-constructor symbol of arity n that is not present in the signature ($S_\mathcal{T}$). The transformation also adds the type-constructor symbol κ to the signature ($S_\mathcal{T}$) and adds for each k in α's type-class constraint K_α a $\forall_\tau\ \alpha_1{:}K_1,\ \ldots,\ \alpha_n{:}K_n.\ \kappa(\alpha_1,\ \ldots,\ \alpha_n) : k$ type-class declaration (to $\mathcal{T}$).

Term Skolemization

The Skolemization for polymorphic terms is similar to the untyped one, except that the appropriate declarations need to be added.

Definition 3.1.5.2 (Term Skolemization). The Skolem transformation for terms ($\Rightarrow_S$) adapted to our polymorphic setting is:

$$\begin{gathered}\forall_\tau \alpha_1{:}K_1,\ \dots,\ \forall_\tau \alpha_n{:}K_n.\ \forall u_1{:}\tau_1,\ \dots,\ \forall u_m{:}\tau_m.\ \ \exists u : \tau.\phi \\ \Rightarrow_S \\ \forall_\tau \alpha_1{:}K_1,\ \dots,\ \forall_\tau \alpha_n{:}K_n.\ \forall u_1{:}\tau_1,\ \dots,\ \forall u_m{:}\tau_m.\ \ \phi\{u \mapsto f\langle \alpha_1,\ \dots,\ \alpha_n\rangle(u_1,\ \dots,\ u_m)\}\end{gathered}$$

where $m \geq 0$, $n \geq 0$ and f is a function symbol of type arity n and term arity m that is not present in the signature ($S_{\mathcal{F}}$). The transformation also adds the function symbol f to the signature ($S_{\mathcal{F}}$) and adds the adequate ($\forall_\tau\ \alpha_1{:}K_1,\ \dots,\ \alpha_n{:}K_n.\tau_1,\ \dots,\ \tau_m \to \tau$) function declaration (to $\mathcal{F}$).

While I chose the type Skolemization to be not satisfiability preserving, the term Skolemization is satisfiability preserving.

Lemma 3.1.5.3. A formula ϕ is satisfiable with respect to a signature Σ if and only if the formula produced by the Skolem transformation $\phi \Rightarrow_S \phi'$ is satisfiable in the extended signature.

Proof. Suppose the lemma does not hold. Then there must be set of values for $\alpha_1,\ \dots,\ \alpha_n$ and $u_1,\ \dots,\ u_m$, where the existential quantifier is satisfiable, but there is no interpretation in the extended signature such that ϕ' is satisfiable. This means that there is an interpretation $\mathcal{I}$ such that $\theta = \{\alpha_1 \mapsto d_1,\ \dots,\ \alpha_n \mapsto d_n\}$ and $\xi = \{u_1 \mapsto e_1,\ \dots,\ u_m \mapsto e_m\}$ for some domains $d_1,\ \dots,\ d_n$ and some elements $e_1,\ \dots,\ e_m$ and $\mathcal{I}_{\theta,\xi}(\exists u : \tau.\ \phi)$ is true and thus $\max\limits_{e \in \mathcal{I}_\theta(\tau)} (\mathcal{I}_{\theta,\xi[u\ \mapsto\ e]}(\phi))$ is true for some $e \in \mathcal{I}_\theta(\tau)$, but there is no interpretation $\mathcal{I}'$ such that $\mathcal{I}'_{\theta,\xi}(\phi\{u \mapsto f\langle\alpha_1,\ \dots,\ \alpha_n\rangle(u_1,\ \dots,\ u_m)\})$ is satisfiable in the extended signature. Clearly, there is one interpretation $\mathcal{I}''$ where $f^{\mathcal{I}}\langle d_1,\ \dots,\ d_n\rangle(e_1,\ \dots,\ e_m) = e$ and thus there must be a $\mathcal{I}''_{\theta,\xi}(\phi\{u \mapsto f\langle\alpha_1,\ \dots,\ \alpha_n\rangle(u_1,\ \dots,\ u_m)\})$ that is true. □

3.2. Orderings

In this section I introduce a typed version of the Knuth-Bendix Ordering, an ordering which is often used by superposition-based automated theorem provers. Orderings are useful, because superposition only requires inferences that fulfill certain ordering constraints. Thus superposition requires an ordering as a parameter. The untyped superposition calculus requires that the ordering is a reduction ordering, which is total on ground terms. The typed superposition calculus, on the other hand, requires a simplification ordering which is total on ground terms.

The commonly used orderings are simplification ordering, therefore requiring a simplification ordering instead of a reduction ordering is not a major restriction. Nonetheless, I will first discuss why the change is required (Sect. 3.2.1). Afterwards, I introduce the Typed Knuth-Bendix Ordering (Sect. 3.2.2).

3.2.1. Simplification Ordering Required

While it is generally known that a total reduction ordering is a simplification ordering in the untyped case, in the type case a total reduction ordering is not necessarily a simplification order. The untyped refutational completeness proof of superposition assumes the fact that a total reduction ordering is a simplification ordering. The fact is used to show that the rewrite systems that are constructed in the candidate interpretation construction are convergent.

First, I sketch why an untyped total reduction ordering is a simplifications ordering, then I show why replacing the reduction ordering by a simplification ordering solves the problem.

Lemma 3.2.1.1 (An untyped total reduction ordering is a simplification order)**.**
If the signature is untyped, a reduction ordering that is total is a simplification ordering, i.e. subterms are smaller than superterms.

Proof Sketch. Suppose there is a subterm that is larger than a superterm. Then we can indefinitely add the context of the subterm to the superterm gaining further smaller terms (because of totality and compatibility with Σ-operations). This cannot be well-founded and thus a total reduction ordering must have the subterm property. □

Unfortunately, a total typed reduction ordering is not necessarily a simplification ordering, because the type of the superterm and subterm can be different and thus they might not share any common context at all. To guarantee completeness in a typed setting my superposition calculus requires a reduction ordering that is a simplification ordering on ground terms instead of a reduction ordering that is total on ground terms. With this change, the rewriting systems (R_C and R_∞) generated by Candidate Interpretation Construction (Sect. 2.2.4) are still confluent and Lemma 2.2.4.2 can be restated as:

Lemma 3.2.1.2.
The rewrite systems R_C and R_∞ are convergent.

Proof. Obviously, $s \succ t$ for all rules $s \to t$ in R_C and R_∞. Furthermore, it is easy to check that there are no critical pairs between any two rules: Assume that there are rules $u \to v$ in E_D and $s \to t$ in E_C, such that u is a subterm of s. As $\succ$ is, on ground terms, a total simplification order, we get $u \prec s$ and therefore $D \prec_C C$ and $E_D \subseteq R_C$. But then s would be reducible by R_C, contradicting condition (6) of the Candidate Interpretation Construction. □

To derive a (ground) typed simplification ordering, it does not seem possible to relax any of the properties of simplification orderings (with the exception that only well-typed substitutions are relevant). In particular, it is not possible to restrict the subterm property to terms of the same type, because then terms could become reducible after they were considered in the model construction. This would make the presented proof of the model construction fail.

3.2.2. A Polymorphically Typed Knuth-Bendix Ordering

The most commonly used orderings by superposition provers are the Knuth-Bendix Ordering (KBO) [3, Page 124] and the Lexicographic Path Ordering (LPO [3, Page 118]). Since our goal is a complete superposition prover, including all machinery, we present an extension of KBO which is designed for our polymorphic language. We extend the standard KBO for terms by introducing a nested KBO for types and show that the adapted version is a simplification order which is total on ground terms. Type classes do not have to be considered separately, since distinct variables must be incomparable and the same variable can only have one type-class constraint.

The main goal of the Typed Knuth-Bendix Ordering, besides having a typed ordering, is to behave as much as possible like the untyped versions, i.e. for the types to have as little impact on the orderings as possible. The motivation for this goal is to keep the configurable simplification heuristics that we developed [9] (in work not included in this dissertation) easily adaptable to the typed setting.

Let w_{f} be a function that maps function symbols to positive numbers, let w_{κ} be a function that maps type-constructor symbols to positive numbers and w their extension to (type) terms (where the weight of terms may or may not include the weight of their types), let $>_{\kappa}$ be a strict ordering on type constructor symbols, $>_{\mathrm{f}}$ be a strict ordering on function symbols and $>_c$ be the transitive closure of the subclass declarations ($\mathcal{TC}$). With the following restrictions, we will call any weight function that fulfills these restriction *admissible*:

1. There exists $w_0 \in \mathbb{R}^+ \backslash \{0\}$ such that for all term variables u, $w(u) = w_0$ for all term constants f, $w(f) \geq w_0$
2. There exists $w_1 \in \mathbb{R}^+ \backslash \{0\}$ such that for all type variables α, $w(\alpha) = w_1$ for all type constants κ, $w(\kappa) \geq w_1$
3. If κ is a unary type constructor symbol with weight $w(\kappa) = 0$, then κ is the greatest element with respect to $>_{\kappa}$
4. If f is a unary function symbol with weight $w(f) = 0$, then f is the greatest element with respect to $>_{\mathrm{f}}$

The type ordering is a KBO defined on type terms. We define $\tau_s \succ_{\tau} \tau_t$ to hold if and only if:

For all type variable $\alpha : C$. $|\tau_s|_{\alpha:C} \geq |\tau_t|_{\alpha:C}$ and

1. $w(\tau_s) > w(\tau_t)$, or
2. $w(\tau_s) = w(\tau_t)$, and one of the following:
   a) There exists a unary type constructor symbol κ, a variable α and a positive integer n such that $\tau_s = \kappa^n(\alpha)$ and $\tau_t = \alpha$
   b) There exist type constructor symbols κ_f, κ_g such that $\kappa_f >_{\kappa} \kappa_g$ and $\tau_s = \kappa_f(\tau_{s_1}, \ldots, \tau_{s_n})$ and $\tau_t = \kappa_g(\tau_{t_1}, \ldots, \tau_{t_n})$
   c) There exists a type constructor symbol κ_f and an index i, $1 \leq i \leq n$, such that $\tau_s = \kappa_f(s_1, \ldots, s_n)$, $\tau_t = \kappa_f(\tau_{t_1}, \ldots, \tau_{t_n})$ and $\tau_{s_1} = \tau_{t_1}, \ldots, \tau_{s_{i-1}} = \tau_{t_{i-1}}$ and $\tau_{s_i} \succ_{\tau} \tau_{t_i}$

Let $=_{Term}$ to be equality but only considering non-type (sub)terms. Then $s \succ_{KBO} t$ holds if and only if:

For all term variables u: $|s|_u \geq |t|_u$ and

1. $w(s) > w(t)$, or
2. $w(s) = w(t)$, and one of the following:
   a) There exists a unary function symbol f, a variable u and a positive integer n such that $s = (f\langle\tau_1, \ldots, \tau_m\rangle)^n(u)$ and $t = u$

   b) There exist functions symbols f, g such that $f >_{\mathrm{f}} g$ and
      $s = f\langle\tau_{s_1}, \dots, \tau_{s_m}\rangle(s_1, \dots, s_n)$ and $t = g\langle\tau_{t_1}, \dots, \tau_{t_m}\rangle(t_1, \dots, t_n)$
   c) There exists a function symbol f and an index i, $1 \le i \le n$, such that $s = f\langle\tau_{s_1}, \dots, \tau_{s_m}\rangle(s_1, \dots, s_n)$,
      $t = f\langle\tau_{t_1}, \dots, \tau_{t_m}\rangle(t_1, \dots, t_n)$ and
      $s_1 =_{Term} t_1, \dots, s_{i-1} =_{Term} t_{i-1}$ and $s_i \succ_{KBO} t_i$

and $s \succ_{TKBO} t$ to hold if and only if:

1. $s \succ_{KBO} t$, or
2. $s =_{Term} t$, and there exists a function symbol f such that $s = f\langle\tau_{s_1}, \dots, \tau_{s_m}\rangle(s_1, \dots, s_n)$ and $t = f\langle\tau_{t_1}, \dots, \tau_{t_m}\rangle(t_1, \dots, t_n))$ and
   a) an index i, $1 \le i \le m$ such that $\tau_{s_1} = \tau_{t_1}, \dots, \tau_{s_{i-1}} \succ_\tau \tau_{t_{i-1}}$ and $\tau_{s_1} \succ_\tau \tau_{t_i}$, or
   b) for all index i, $1 \le i \le m$ $\tau_{s_i} = \tau_{t_i}$ and an there exists an index j, $1 \le j \le n$, such that $s_1 = t_1, \dots, s_{j-1} = t_{j-1}$ and $s_j \succ_{TKBO} t_j$

It is obvious that $\succ_\tau$ and $\succ_{KBO}$ are simplification orderings, because they are essentially restating KBO once for type terms and once of typed terms where the types are ignored. I now show that their combination into the Typed Knuth Bendix Ordering $\succ_{TKBO}$ results in a simplification order. The Typed Knuth Bendix Ordering tries to emulate KBO as much as possible, only resorting to using the types to break ties.

Here I show that the Typed Knuth Bendix Ordering is a simplification ordering by first showing that it is a reduction ordering and then showing the simplification ordering.

Lemma 3.2.2.1 (Typed Knuth-Bendix Ordering is a Reduction Ordering).
Let w_{f} be an admissible weight function for $>_{\mathrm{f}}$ then the Typed Knuth-Bendix Ordering ($\succ_{TKBO}$) induced by $>_\kappa$ and $\succ_\tau$ is a reduction ordering which is total on ground terms.

Proof. The usual proof (see [3, Page 112]) by using simplification orderings is not sufficient. It fails because the requirement is on a combination of subterms, but we cannot use all these subterms, because we cannot guarantee well-typedness. To show that $\succ_{TKBO}$ is a reduction ordering, we have to show all properties of a reduction ordering (Section 2.2.2):

1. It has to be total on ground terms s_1, s_2: $s_1 \succ_{TKBO} s_2 \lor s_2 \succ_{TKBO} s_1 \lor s_1 = s_2$

   If $s_1 \neq s_2$ then we have two cases. Either $s_1 \neq_{Term} s_2$, then we use TKBO case 1 and use the ordering $\succ_{KBO}$, which is total on ground terms, to order s_1 and s_2. The other case is $s_1 =_{Term} s_2$ but $s_1 \neq s_2$, then there is a position at which a type of s_1 and the corresponding type of s_2 are different and thus $\succ_{TKBO}$ case 2b applies until such a position is reached. Then $\succ_{TKBO}$ case 2a uses $\succ_\tau$, which is total on ground type terms.

2. It has to be irreflexive for all terms s: $\neg(s \succ_{TKBO} s)$

   We argue by induction over s. The case where s is a variable is trivial, since there is no rule in $\succ_{TKBO}$, $\succ_{KBO}$ or $\succ_\tau$ to make any variable larger than a variable, thus they cannot be reflexive. In the case where s is function term ($s = f\langle\tau_{s_1}, \dots, \tau_{s_m}\rangle(t_1, \dots, t_n)$). For $\succ_{TKBO}$ case 1 we know that $\succ_{KBO}$ is irreflexive and are done. In $\succ_{TKBO}$ case 2a we know that $\succ_\tau$ is irreflexive and are done. Finally for $\succ_{TKBO}$ case 2b we can apply the induction hypothesis and are done.

3. It has to be transitive for all s_1, s_2, s_3: $(s_1 \succ_{TKBO} s_2 \wedge s_2 \succ_{TKBO} s_3) \implies (s_1 \succ_{TKBO} s_3)$

   From $s_1 \succ_{TKBO} s_2$ and $s_2 \succ_{TKBO} s_3$ we know that $|s_1|_u \geq |s_2|_u \geq |s_3|_u$ and thus the variable conditions for $\succ_{KBO}$ are fulfilled. We also know that $w(s_1) \geq w(s_2) \geq w(s_3)$. If any of the inequalities is strict, then $w(s_1) > w(s_3)$ and thus $s_1 \succ_{TKBO} s_3$ must hold ($\succ_{TKBO}$ case 1). Thus the case $w(s_1) = w(s_3)$ remains. Now either $s_1 \neq_{Term} s_3$ or $s_1 =_{Term} s_3$. If $s_1 \neq_{Term} s_3$ then $s_1 =_{Term} s_2$ and $s_2 \neq_{Term} s_3$, $s_1 \neq_{Term} s_2$ and $s_2 =_{Term} s_3$ or $s_1 \neq_{Term} s_2$ and $s_2 \neq_{Term} s_3$. In all three of these cases, transitivity of $\succ_{KBO}$ suffices to show transitivity of $\succ_{TKBO}$.

   Let $\tau(t) ::= \tau_1, \ldots, \tau_{last}$ be the sequence of types of a term t by listing them from left to right and let $\tau(t)_i$ be the ith element of that sequence. If $s_1 =_{Term} s_3$ and $s_2 =_{Term} s_3$ then there are positions i_1 and i_2 such that $\tau(s_1)_{i_1} \succ_\tau \tau(s_2)_{i_1}$ and $\tau(s_2)_{i_2} \succ_\tau \tau(s_3)_{i_2}$ and for all $j_1 < i_1$ and $j_2 < i_2$ it holds that $\tau(s_1)_{j_1} = \tau(s_2)_{j_1}$ and $\tau(s_2)_{j_2} = \tau(s_3)_{j_2}$. Then either $i_1 < i_2$ and $\tau(s_1)_{i_1} \succ_\tau \tau(s_2)_{i_1} = \tau(s_3)_{i_1}$ and thus $s_1 \succ_{TKBO} s_3$. Or $i_1 = i_2$ and $\tau(s_1)_{i_1} \succ_\tau \tau(s_2)_{i_1} \succ_\tau \tau(s_3)_{i_1}$ and thus $s_1 \succ_{TKBO} s_3$. Or $i_1 > i_2$ and $\tau(s_1)_{i_2} = \tau(s_2)_{i_2} \succ_\tau \tau(s_3)_{i_2}$ and thus $s_1 \succ_{TKBO} s_3$.

4. It has to be compatible with Σ-operations for all $s_1, s_2 \in$ T(Σ,V) and all $m \geq 0$ and all function symbols f of arity m, $s_1 \succ_{TKBO} s_2$ implies $f\langle\tau_1, \ldots, \tau_m\rangle(t_1, \ldots, s_1, \ldots, t_m) \succ_{TKBO} f\langle\tau_1, \ldots, \tau_m\rangle(t_1, \ldots, s_2, \ldots, t_m)$

   Either $s_1 \neq_{Term} s_2$ or $s_1 \neq s_2$. If $s_1 \neq_{Term} s_2$ then $f(t_1,\ldots,s_1,\ldots,t_m) \neq_{Term} f(t_1,\ldots,s_2,\ldots,t_m)$ and thus the compatibility follows from the compatibility of $\succ_{KBO}$. If $s_1 =_{Term} s_2$ then $f(t_1,\ldots,s_1,\ldots,t_m) =_{Term} f(t_1,\ldots,s_2,\ldots,t_m)$. Also $\tau(f\langle\tau_1, \ldots, \tau_m\rangle(t_1, \ldots, s_1, \ldots, t_m))$ and $\tau(f\langle\tau_1, \ldots, \tau_m\rangle(t_1, \ldots, s_2, \ldots, t_m))$ are identical except (possibly) for the subsequences $\tau(s_1)$ respectively $\tau(s_2)$ and therefore $s_1 \succ_{TKBO} s_2$ implies $f\langle\tau_1, \ldots, \tau_m\rangle(t_1, \ldots, s_1, \ldots, t_m)$ $\succ_{TKBO} f\langle\tau_1, \ldots, \tau_m\rangle(t_1, \ldots, s_2, \ldots, t_m)$.

5. It has to be closed under substitutions for all $s_1, s_2 \in$ T(Σ,V) and all substitutions σ $s_1 \succ_{TKBO} s_2$ implies $s_1\sigma \succ_{TKBO} s_2\sigma$

   Either $s_1 \neq_{Term} s_2$ or $s_1 =_{Term} s_2$. If $s_1 \neq_{Term} s_2$ then closed under substitutions follows from the fact that $\succ_{KBO}$ is closed under substitutions and that $s_1\sigma \succ_{KBO} s_2\sigma$ implies that $s_1\sigma \neq_{Term} s_2\sigma$. If $s_1 =_{Term} s_2$ then there is a position i such that $\tau(s_1)_i \succ_\tau \tau(s_2)_i$ and for all $j < i$ it holds that $\tau(s_1)_j = \tau(s_2)_j$. Then closed under substitutions for $\succ_\tau$ means that $\tau(s_1)_i\sigma \succ_\tau \tau(s_2)_i\sigma$ and for all $j < i$ it holds that $\tau(s_1)_j\sigma = \tau(s_2)_j\sigma$. Therefore, and because $=_{Term}$ is also closed under substitutions it holds that $\tau(s_1\sigma)_i \succ_\tau \tau(s_2\sigma)_i$ and for all $j < i$ it holds that $\tau(s_1\sigma)_j = \tau(s_2\sigma)_j$. Thus $s_1\sigma \succ_{KBO} s_2\sigma$ holds.

□

Now, the only property that remains to be shown is the subterm property, then the Typed Knuth-Bendix Ordering is also a simplification ordering.

Lemma 3.2.2.2 (Typed Knuth-Bendix Ordering is a Simplification Ordering)**.**
Let w_{f} be an admissible weight function for $>_{\mathrm{f}}$ then the Typed Knuth-Bendix Ordering ($\succ_{TKBO}$) induced by $>_{\kappa}$ and $\succ_\tau$ is a simplification ordering which is total on ground terms.

Proof. A simplification ordering is a reduction ordering with the subterm property.

1. The ordering is a reduction ordering (Lemma 3.2.2.1)

2. If $s[t]$ and $s \neq t$ then $s \succ_{TKBO} t$.

   If t is a proper subterm of s then $s \neq_{Term} t$. Since the subterm property holds for $\succ_{KBO}$ it holds for $\succ_{TKBO}$.

□

3.3. The Superposition Calculus

I now present the superposition calculus for the polymorphic first-order language with type classes.

Positive Superposition
only if conditions 2–8 hold

$$\frac{D' \lor t \approx t' \quad C' \lor s[s_2]_p \approx s'}{(D' \lor C' \lor s[t']_p \approx s')\sigma} \ (PSup)$$

Equality Factoring
only if conditions 1–3 and 10 hold

$$\frac{C' \lor s \approx s' \lor t \approx t'}{(C' \lor t \approx s' \lor t' \not\approx s')\sigma} \ (EF)$$

Negative Superposition
only if conditions 2–6 and 9 hold

$$\frac{D' \lor t \approx t' \quad C' \lor s[s_2]_p \not\approx s'}{(D' \lor C' \lor s[t']_p \not\approx s')\sigma} \ (NSup)$$

Equality Resolution
only if conditions 9 and 11 hold

$$\frac{C' \lor s \not\approx s'}{C'\sigma} \ (ER)$$

Let $\prec$ be a fixed simplification order that is total on ground terms. I refer to $s[s_2]_p$ also as s.

1. σ is the mgu of s and t 2. $s\sigma \not\preceq s'\sigma$ 3. $t\sigma \not\preceq t'\sigma$ 4. s_2 is not a term variable

5. $(t \approx t')\sigma$ is *strictly maximal* in $(D' \lor t \approx t')\sigma$, nothing selected 6. σ is the mgu of t and s_2

7. $(s \approx s')\sigma$ is *strictly maximal* in $(C' \lor s \approx s')\sigma$, nothing selected 8. $t\sigma \approx t'\sigma \not\succeq s\sigma \approx s'\sigma$

9. $((s \not\approx s')\sigma$ is *maximal* in $(C' \lor s \not\approx s')\sigma$, nothing selected$) \lor s \not\approx s'$ selected

10. $(t \approx t')\sigma$ is *maximal* in $(C' \lor s \approx s' \lor t \approx t')\sigma$, nothing selected 11. σ is the mgu of s and s'

As we can see, the side conditions and the inference rules themselves are identical to the untyped case (Sect. 2.2.3). In fact, the polymorphic language was designed such that only the machinery of superposition has to be adapted, not the inference rules themselves. What has changed is the definition of clauses, terms and their notation $[]_p$ to access a subterm and the (most general) unification. The restriction on the orderings is changed from reduction ordering to simplification ordering.

In the completeness proof below, we show that superposition calculus is still refutationally complete (Section 3.5). We will use the same inference rules and introduce intermediate languages to lift completeness from ground superposition, via the intermediate languages, to the polymorphic language.

3.4. Implementation and Evaluation

In this section, I provide evidence that the native implementation of type systems into the superposition calculus is superior to (efficient) type-encodings into untyped first-order logic. I further show that native support for the polymorphism extended with type classes is competitive with monomorphisation [11], an incomplete but very efficient native encoding. For problems with thousands of axioms, the polymorphic type system (with type classes) outperforms monomorphisation.

3.4.1. The Pirate Implementation

I have implemented the polymorphic calculus with type classes in a tool called *Pirate*. Pirate is written in (about 25,000 lines of) Scala. The CNF transformation implemented by Pirate is rather naive. The only improvement implemented is polarity-based replacement of equivalences. Furthermore, Pirate does not support splitting, but it supports many common redundancy elimination rules [67], such as Subsumption, Rewriting, Merging Replacement Resolution, Assignment Equation Deletion, Condensation and Tautology removal. Its main indexing is based on Path Indexing [60], while Subsumption uses a variant of Feature Vector Indexing [57]. Pirate uses a global hash-based term sharing that guarantees that every term has at most one object representing it (during the complete runtime). It uses a worked-off/usable main-loop with given-clause selection heuristics and ordering computation similar to our extension of SPASS [9]. It has full proof search documentation, i.e. for all clauses Pirate records which inference or reduction introduced or removed them. Pirate has an independent proof checking subsystem, that translates a proof consisting of inferences or reduction rules to a simpler proof that uses only typed-check instantiations and the inferences of the superposition calculus without computing unifiers (i.e. only on syntactically equal terms). The unifiers are separated from inferences to check that all (type) substitutions of a proof are well-typed and withhin the type class constraints.

3.4.2. Setup

I have evaluated my prototype on problems generated by Sledgehammer [10] from Isabelle/HOL theories. Each problem corresponds to one user action within a proof. The theories are of areas as diverse as the fundamental theorem of algebra, the completeness of Hoare logic and the type soundness of a subset of Java. I used the 1249 problems generated from seven of those Isabelle theory files[1] and tested them on seven different type encodings, each with nine different number

[1]I evaluated on Arrow Order, FFT, FTA, Hoare, TypeSafe, StrongNorm and NS Shared [10].

of facts per problem[2]. A fact is either an Isabelle definition or theorem, they are selected and their importance is ranked by Sledgehammer. I used a time limit of 1 hour on one HyperThread of a cluster of 2xIntel Xeon E5620 using a JVM heap size of 6 GB. For the comparison, I used the following type encodings:

Type Classes	Using native polymorphism with type classes.
Poly Native	Using native polymorphism with a predicate-based type class encoding.
Monomorph	An incomplete encoding of polymorphism with type classes by picking relevant monomorphic instances [15]. Native monomorphism is currently the most efficient type encoding used in Isabelle [9, pp. 11-12].
Poly Tags	Encodes polymorphism with type classes into untyped first-order logic with the help of special typing function symbols.
Light Poly Tags	Encodes polymorphism with type classes into untyped first-order logic with a reduced number of special typing function symbols due to monotonicity analysis of the occurring types.
Poly Guards	Encodes polymorphism into untyped first-order logic with the help of special typing predicate symbols.
Light Poly Guards	Encodes polymorphism with type classes into untyped first-order logic with a reduced number of special typing predicate symbols due to monotonicity analysis of the occurring types.

The encodings are performed by Sledgehammer and more details on them can be found in [11]. For the Type Classes encoding, 6 to 9 problems (0.5%-0.7%) have a subclass structure and corresponding type declarations that are not coregular. It is my impression that this is because some intermediate type classes are missing. This means that for the not coregular part of these problems Pirate is not refutational complete. Pirate still processes the problems by computing unification as if the problem is coregular (thereby losing most general unification for those type classes). This approach is sound, because the unifiers are still unifiers just not most general.

3.4.3. Evaluation

In the evaluation we first look at the success rate, which shows that the type class encoding is highly successful. For larger fact sizes the type class encoding beats all other encodings, including the monomorphic encoding. Most likely, the reason for this is the duplication of facts that the monomorphic encoding requires. In general, it also requires fewer inferences than the

[2]16, 32, 64, 128, 256, 512, 1024, 2048 and 4096 facts before translation. Note that Sledgehammer can not reliably generate more than 4096 facts (see also comment to Figure 3.2)

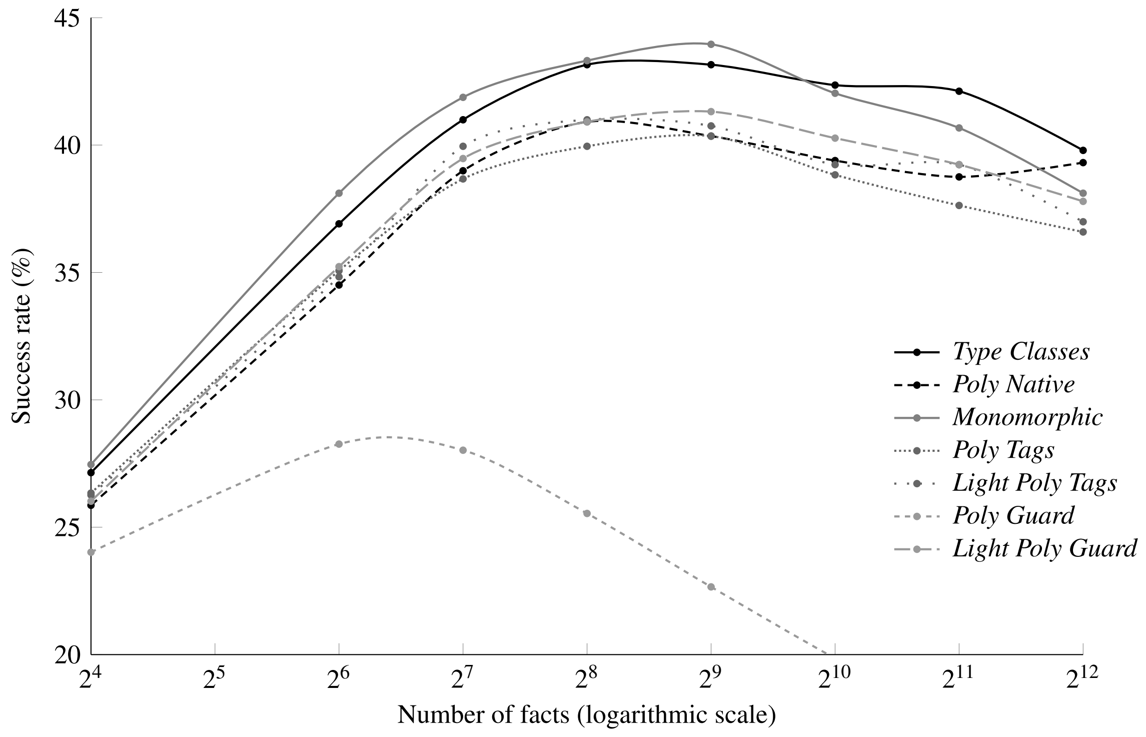


Figure 3.1.: Success rate and scalability of each type encoding

monomorphic encoding, even when its success rate is higher. The native encoding (type classes and monomorphic) are the encodings with the highest success rate. They also have the highest success rate without performing any inferences, i.e. by using just the simplification rules. We will also see that, while Pirate is less tuned than SPASS (e.g. it has no build in decision procedures), its success rate is between 74% to 118% of SPASS's success rate. In the following we will look closer at these and similar results of the evaluation.

Success Rate and Scalability of the Type Encodings

Figure 3.1 shows the success rate and scalability of the tested encodings. The horizontal axis denotes the number of facts in logarithmic scale and the vertical axis the percentage of problems proved in a single run of up to one hour.

The native monomorphic and the native type class encodings are clearly superior to the encodings into untyped first-order logic. From 16 until 512 facts, the monomorphic encoding has the highest success rate and the type class encoding the second highest success rate. From 1024 to 4096 facts, the type class encoding has the highest success rate and the monomorphic encoding the second highest success rate — except for 4096 facts, where the poly native encoding is the second best and the monomorphic encoding only third. The encoding into untyped first-order logic with the highest success rate for 16, 128 and 256 facts is the light poly tags encoding; for the other fact sizes it is the light poly guards encoding.

The number of facts determines the success rate. With few facts, many problems do not have

Number of Facts	Type Classes	Monomorphic	Type Classes versus Monomorphic
16	30	32	92% (−2)
32	50	62	80% (−12)
64	73	120	61% (−47)
128	140	223	63% (−83)
256	271	388	70% (−117)
512	535	685	78% (−150)
1024	1064	1289	83% (−225)
2048	2112	2584	82% (−472)
4096	4083	4979	82% (−896)

Figure 3.2.: Average number of initial formulas: Type Classes versus Monomorphic

all (for the proof) necessary facts available, but those that have all necessary facts are then easier to solve. With hundreds and thousands of facts, the encodings have so many facts available that the success rate starts to decline. This is most likely because the search space becomes too large because of the large number of potentially irrelevant (for a successful proof) facts. The most successful fact set size for poly native, light poly tags and poly guards is 256 facts; for monomorphic, poly tags and light poly guards it is 512 facts, while the type class encoding performs best for both 256 and 512 facts.

The cause for the different success rates of the various type encoding is difficult to determine for each individual problem. Already slight differences in the clause set (caused by the differences of the encodings) result in different exploration of the search space. It is generally known that any difference has the possibility of dramatic changes in runtime. Nonetheless, the higher success rate of the monomorphic encoding can be traced (at least in part) to a simpler problem structure. The monomorphic encoding only uses some (heuristically picked) ground instances of the types the type class and poly native encoding have to consider. One quantifiable indication of this is the number of saturations that can be found for 16 facts. The monomorphic encoding can saturate 26% of the problems, significantly more than the type class encoding (17% saturated) and the poly native encoding (14% saturated).

The downside of the simpler structure of monomorphic problems is that it duplicates facts (for each ground type instance the fact is either duplicated or excluded). So a formula in the type class or poly native encoding can become several formulas in the monomorphic encoding, one for each picked type instance. Figure 3.2 shows the number of initial formulas for both the monomorphic and type class encoding. For 4096 facts, an initial monomorphic problem has an average of 900 formulas more than the corresponding type class problem, while for 16 facts the average difference is just 2 formulas. The addition of a handful of duplicates has in practice a low impact, because the solver can still process them efficiently. Adding several hundred of duplicates on the other hand can significantly slow down the solver. In fact, 90% (= 293 proofs) of type class proofs for 4096 facts are found with picking (less than) 900 clauses for performing inferences — and each formula can result in multiple clauses. In comparison, 90% (= 276 proofs) of the monomorphic proofs for 4096 facts are found with picking (less than) 1400 clauses and 293 proofs are found with picking (less than) 2500 clauses for performing inferences.

Note that Sledgehammer cannot generate more than 4096 facts reliably. This can already be seen at 4096 facts, where the average number of formulas for the type class encoding is already below target (with generating only 4083 formulas). Even though there should be more than 4096 formulas due to the helper formulas of encoding higher-order features (as seen in all other fact sizes, where there are more formulas than facts).

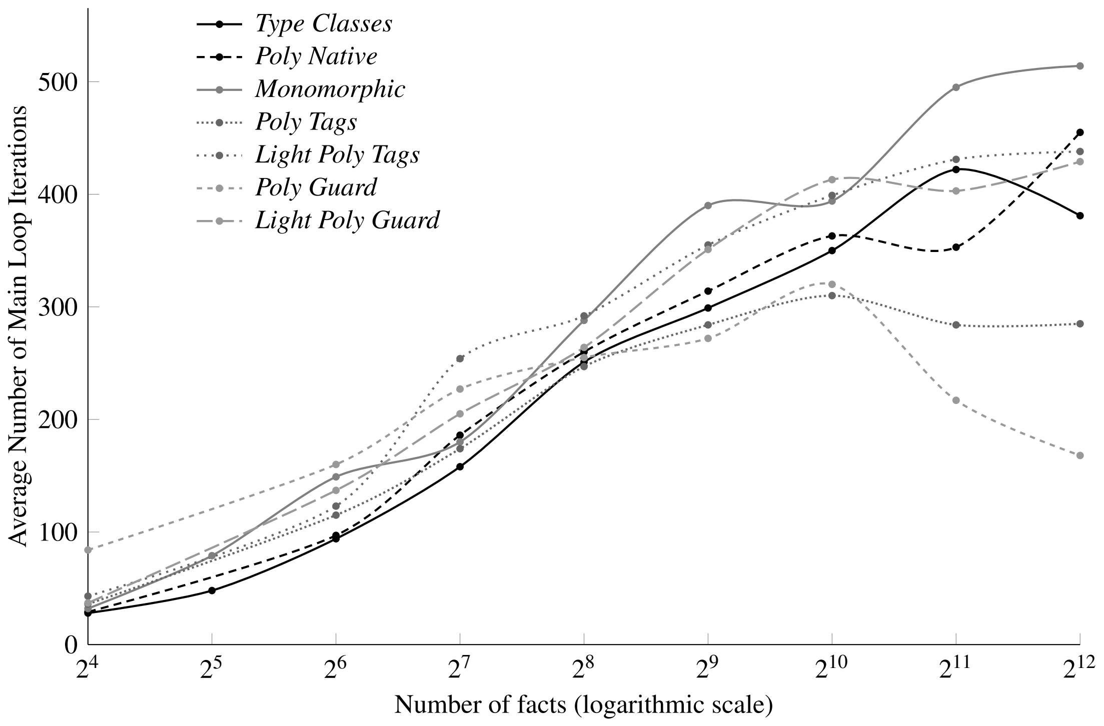


Figure 3.3.: Average number of main loop iterations per proof

The average number of clauses picked for performing inferences with (for each encoding and fact size) is given in figure 3.3. It shows that the monomorphic encoding requires more inferences than the type class encoding. This is even the case when it proves fewer problems (i.e. see the results from 1024 to 4096 facts). The type class encoding generally also requires fewer inferences than the encodings into untyped first-order logic. The exceptions are only at fact sizes where the encodings into untyped first-order logic have lower success rate. The lower success rate is from 3 percentage points (poly tags at 256 to 1024 facts and light poly guards at 2048 facts) up to 23 percentage points (poly guards at 4096 facts).

Variance Inbetween Fact Sizes

Figure 3.4 shows, for each type encoding, the percentage of problems that can be proven by at least *N* different fact sizes. Experiments with SPASS shows similar results (SPASS does not support the poly native and type classes encodings though). The type class encoding has the most problems that can be solved by all fact sizes. All encodings have a number of problems that can be proven only by few of the fact sizes. Around twice as many problems can be proven by one or more fact sizes compared to the number of problems proven by all fact sizes. The reasons include that the smaller fact sizes tend to not include all necessary facts, in particular for

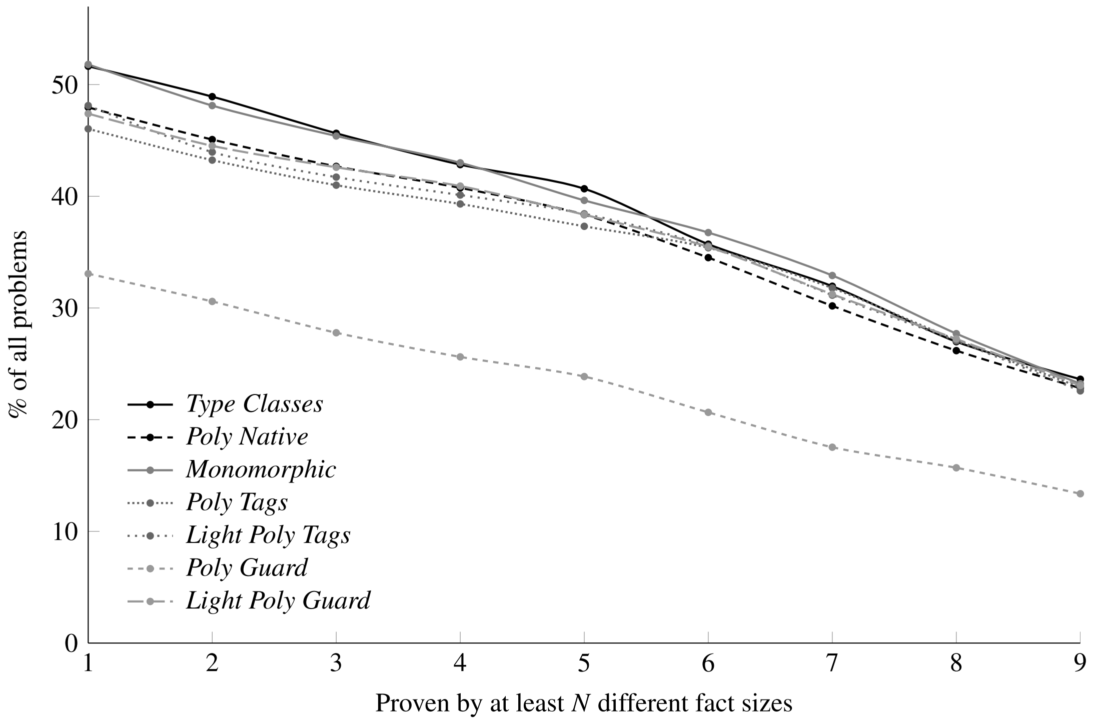


Figure 3.4.: Stability of the type encodings

16 facts, hundreds of saturation can be found. Whereas larger fact sizes overwhelm the solver by increasing the search space and making proofs found in smaller problems harder to find.

Proofs Using Only Simplifications

Figure 3.5 shows, for each type encoding, the percentage of problems that can be proven only with simplifications (i.e. without performing any inferences). Interestingly, there is a clear partition in three parts. The least successful encoding is the poly guards encoding with a success rate below 5%. The monomorphic and the type classes encoding have the highest success rate. Their success rate is between 11% (16 facts) and 14% (from 1024 facts on). Inbetween are all other encodings with a success rate between 8% (poly tags at 16 facts) and 11% (poly native at 4096 facts). A possible reason for the difference between the native encodings and the encodings into untyped logic is that all non-native encodings have to encode the type classes. The type class encoding provides native support while the monomorphic encoding removes them by just considering some type instances.

Time Until Successful Proof

Here, I compare the type class encodings average time until it find proofs with the light poly guards encoding and then with the monomorphic encoding.

There are multiple possible ways to reasonably measure the average time. My goal is to avoid penalizing the encoding that is more successful, because (in general) harder proofs require more

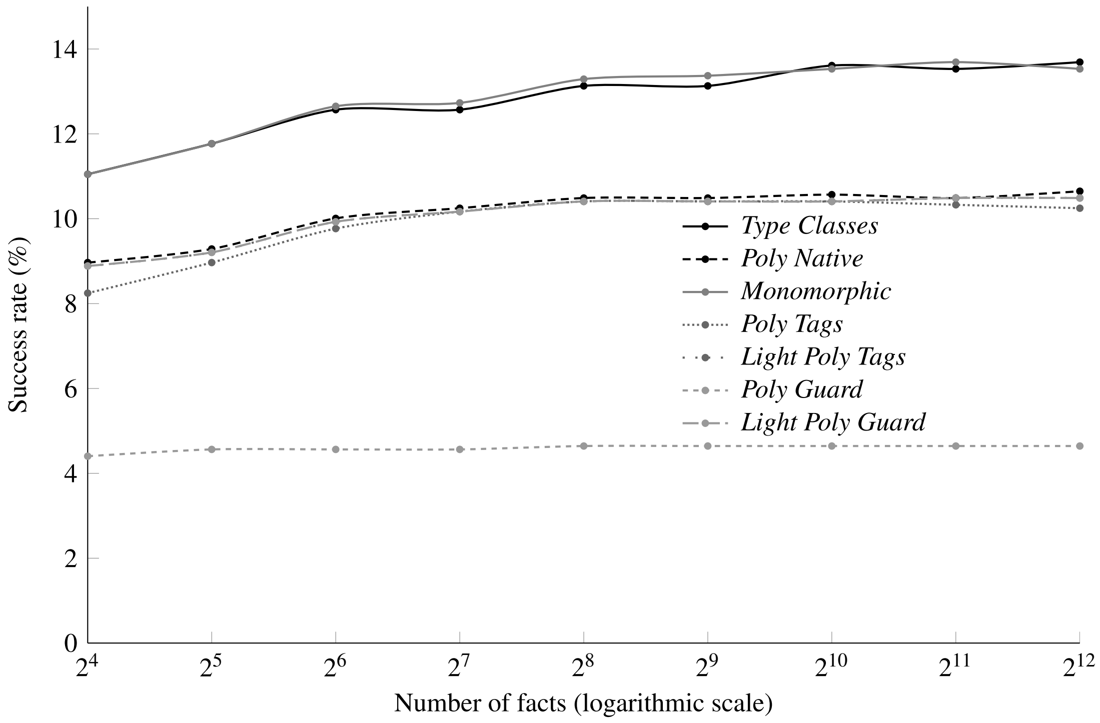


Figure 3.5.: Provable by Simplifications Only

	16	64	128	256	512	1024	2048	4096
Type Classes versus Light Poly Guards								
Both Proved	319	421	471	483	476	464	454	430
Max Proofs	325	440	493	511	516	503	490	472
Type Classes versus Monomorphic								
Both Proved	338	454	499	511	508	487	476	436
Max Proofs	339	461	512	539	539	525	508	476

Figure 3.6.: Number of proofs used for comparison of average proof time

time. We can expect that encodings with higher success rate also prove more of the harder problems and thus take longer. I chose two ways to compare the average time, which are both suitable for comparison and additionally invariant to the length of the timeout: First, the average time until a proof was found for those problems where both encodings find a proof (marked as Both Proved). The second is the average time to find the N fastest proofs, where N is the number of proofs the worse of the two encodings finds (marked as Max Proofs). It is possible that the N fastest proofs of the two encodings are from different problems. Note that a comparison using those averages should only be done within one fact size and figure, because the averages of different fact sizes are based on a different number of potentially different problems. Figure 3.6 presents the number of problems used at each fact size for both averages. Figure 3.7 shows the two average times for the type class and light poly guards encoding. Except for the Both Proved

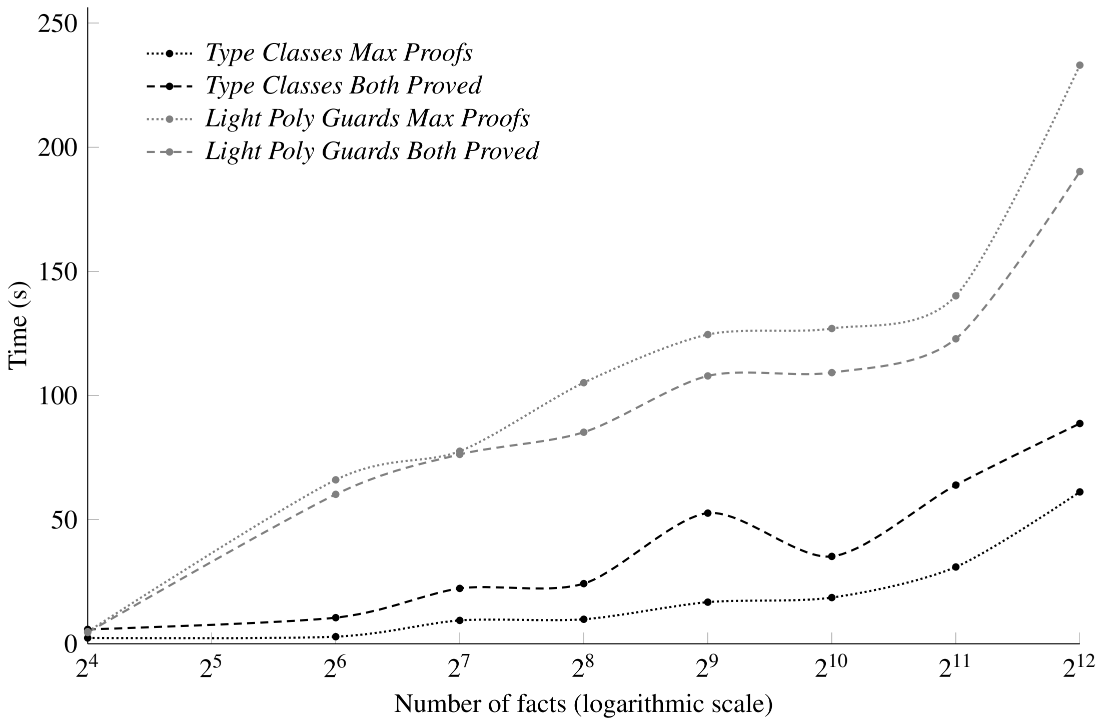


Figure 3.7.: Average time until proof: Type Classes versus Light Poly Guards

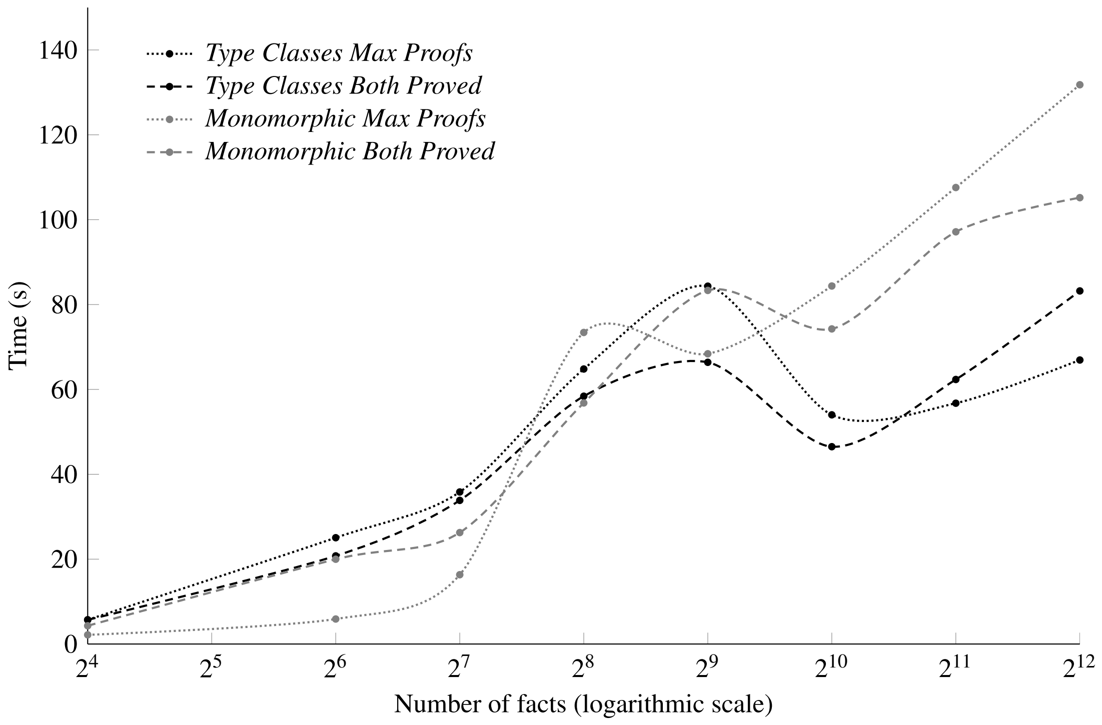


Figure 3.8.: Average time until proof: Type Classes versus Monomorphic

average for 16 facts the type class encoding is always faster than the light poly guards encoding. It also consistently solves more problems.

Figure 3.8 shows the two average times for the type class and monomorphic encoding. For smaller fact sizes (up to 128 facts), the monomorphic encoding is faster and solves more problems. For larger fact sizes, the type class encoding is faster and solves more problems (from 1024 facts).

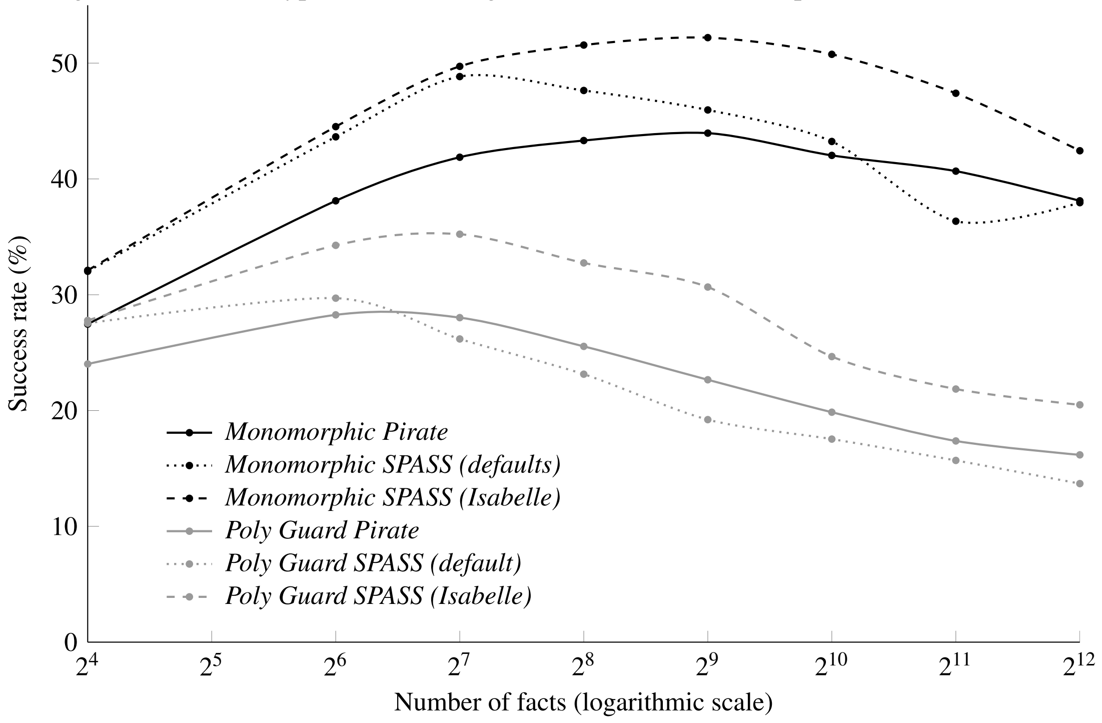


Figure 3.9.: Success rate SPASS versus Pirate

Comparison between Pirate and SPASS

For reference, I have compared the performance of our prototype with the performance of the SPASS version 3.8ds which we specifically optimized for Isabelle/HOL [9] (marked Isabelle) and the same SPASS version with default settings (marked defaults). The SPASS binary was created using profile guided whole program optimizations on similar Isabelle problems.

I used the monomorphic and the poly guards type encoding for the comparison of SPASS and Pirate. The monomorphic encoding is the only native encoding SPASS supports while the poly guards encoding is Pirate's lowest performing (i.e. most difficult) type encoding. Additionally, SPASS has special support for the poly guards' predicates in form of its support for sorts. The main difficulty of the poly guards encoding is rewriting, because the guards added to previously unconditional equations make unconditional rewriting impossible.

The success rates of Pirate and SPASS on the two encodings are shown in Figure 3.9. Pirate and SPASS's Isabelle setting show similar scalability, but Pirate has a lower success rate. Pirate's monomorphic success rate is between 83% (1024 facts) and 90% (4096 facts) of SPASS's Isabelle setting, while Pirate's poly guards success rate is between 74% (512 facts) and 86% (16 facts) of SPASS's Isabelle setting. Pirate's monomorphic success rate is between 86% (16 and 128 facts) and 112% (2048 facts) of SPASS's default setting, while Pirate's poly guards success rate is between 87% (16 facts) and 118% (512 and 4096 facts) of SPASS's default setting.

I believe that the main causes of the difference between SPASS and Pirate is, that Pirate's CNF transformation is naive, SPASS's reductions are more powerful and its algorithms better tuned. Furthermore, SPASS is circa one order of magnitude faster, which is might be caused by a combination of the above and the fact that SPASS is written in hand-tuned C, while Pirate is written in Scala and runs on the Java Virtual Machine.

The evaluation results are available under

`http://people.mpi-inf.mpg.de/~dwand/thesis/`

3.5. Refutational Completeness

The refutational completeness proof for polymorphic superposition with type classes is a composition of four liftings, four encodings and five (variants of) first-order languages. Each lifting step starts from a first-order language for which we know that superposition is refutationally complete and uses that to prove completeness for the first-order language with an extended type system. The lifting steps require the more expressive language to be encoded into the language we lift from. The encoding step must preserve satisfiability.

The initial language is the standard first-order language with **no types**, in particular its **ground** version for which superposition is known to be refutationally complete (Sect. 2). It is represented by the bottom-most box in Figure 3.10. The first step then lifts completeness from superposition for ground untyped first-order to **ground** first-order with **type symbols**, represented by the second box from the bottom (Sect. 3.5.1). The second lifting then further extends completeness to the middle box of the figure — **non-ground** first-order with **type symbols** (Sect. 3.5.2). Thirdly, completeness of **non-ground** first-order with type-symbols is lifted to **monomorphic** first-order represented by the forth box from the bottom — a non-ground first-order language with type terms but no type variables (Sect. 3.5.3). Finally, the completeness for monomorphic first-order is lifted to **polymorphic** first-order with **type classes**, represented by the top box (Sect. 3.5.4).

The proof obligations for each step are as follows. First, an encoding from the more expressive to the previous logic has to be presented. That encoding has to be satisfiability preserving. To complete each step, we also have to lift the inferences to the more expressive logic. To proof the Model Construction Lemma (Lemma 2.2.4.8) using the Candidate Interpretation Construction (Section 2.2.4) it suffices to show the Lifting Lemmas for each inference of the superposition calculus (Lemma 2.2.4.6 and Lemma 2.2.4.7) as well as showing variables closed under rewriting in the Candidate Interpretation Construction (Lemma 2.2.4.5). Once these three lemmas are proven, the Model Construction Lemma follows and therefore static refutational completeness (Lemma 2.2.4.11) does too. For the intermediate lifting, we will not show dynamic refutational completeness, which is only required to run or meaningfully implement superposition. We will show dynamic refutational completeness for the last lifting to the polymorphic language with type classes.

The superposition calculus works on clauses. I have already shown that clausification and Skolemization are possible and can be used to transform formulas of the presented polymorphic language into clauses. Therefore, I define the intermediate languages only for clauses and not for formulas. Furthermore, to obtain a more compact presentation, I omit predicates. The rules for predicates are also very similar to those of functions. In fact, in a typed setting, predicates could be encoded as functions of a boolean type.

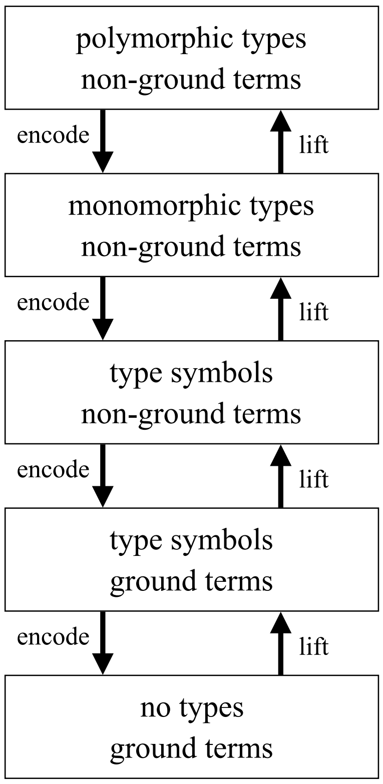


Figure 3.10.: Completeness proof

In Section 3.5.1 I will first present the syntax and semantics of the type-symbol first-order language and then its completeness proof for ground terms. This is followed by the completeness proof for non-ground terms (Sect. 3.5.2). Then, I present the syntax and semantics of the monomorphic first-order language and its proof (Sect. 3.5.3). Finally, I present the refutational completeness proof for the already presented polymorphic language with type classes (Sect. 3.5.4).

3.5.1. Lifting to Type Symbols without Term Variables

Here, I present the first typed first-order intermediate language. The language has terms with variables and a type system of type symbols which is in some respects similar to type constants in the polymorphic language. The type symbols are essentially sorts of a many-sorted system, but more general type declarations for function symbols are allowed. In particular, overloaded return types are possible in the type-symbol language. The language is not quite an instance of the polymorphic language presented above, because the type declaration of functions are more general than an instance derived by restricting the complex polymorphic type terms to be just type constants. I chose this route, because I believe it leads to a simpler presentation and most of the proofs than trying to achieve the same with an instance would have. When showing refutational completeness, I will first show the ground case and untyped ground case to be equivalent and then lift the type-symbol ground case to variables with type-symbols. Complex ground type terms present in the polymorphic calculus can be represented by a fresh type-symbol. We will use this in our lifting from type-symbols to monomorphic first-order logic.

Syntax

The syntax of our type-symbol language is the first step away from untyped first-order logic. For the ground case, the type-symbol language and the untyped language are essentially the same and we will show this below. In contrast to the polymorphic language, we define well-typedness and the terms and clauses simultaneously, since the type-symbol case is simpler. We therefore only define well-typed type-symbol terms and clauses.

Definition 3.5.1.1 (Type-Symbol Signature). The signature Σ^T for type-symbol first-order logic is the tuple $(S^T_{\mathcal{F}}, S^T_{\mathcal{T}}, \mathcal{F}^T)$ where

- $S^T_{\mathcal{F}}$ is a countable set of function symbols f with arity ≥ 0, written as $\text{arity}(f) = m$;
- $S^T_{\mathcal{T}}$ is a countable set of type symbols. We will use κ for type symbols in this section;
- $\mathcal{F}^T$ is a set of tuples $(f, \kappa_1,\ldots,\kappa_m \rightarrow \kappa, \kappa_{a_1}, \ldots, \kappa_{a_n})$ where f is a function symbol contained in $S^T_{\mathcal{F}}$ with arity $m \geq 0$, symbol-arity $n \geq 0$ and where κ is the type symbol that represent the range and $\kappa_1,\ldots,\kappa_m$ are the type symbols for the respective arguments domain. For each pair of function symbol f and type symbol κ at most one tuple may exist. Each instance of an overloaded function symbol represents its own function.

  Conceptually, $\kappa_1,\ldots,\kappa_m \rightarrow \kappa, \kappa_{a_1}, \ldots, \kappa_{a_n}$ is a single label τ with two associated functions. One function that maps each τ to a κ that gives its domain and one function that maps each τ to the list of $\kappa_1,\ldots,\kappa_m$ the argument domains. I leave these functions implicit but sometimes write τ instead of $\kappa_1,\ldots,\kappa_m \rightarrow \kappa, \kappa_{a_1}, \ldots, \kappa_{a_n}$.

If a function symbol f has an arity of 0 then we call f a *constant* symbol.

Definition 3.5.1.2 (Type-Symbol Terms). Let Σ^T be a signature and $\mathcal{X}$ be a given countably infinite set of term variables. We split $\mathcal{X}$ for each type symbol κ in $S^T_{\mathcal{T}}$ in disjoint subsets and write $\mathcal{X}_\kappa$ for each countably infinite subset. Then all *type-symbol terms* $T^T_\Sigma(\mathcal{X})$ and their type symbol $\tau(\ldots)$ are recursively defined as follows:

1. For all sets of variables $\mathcal{X}_\kappa$ each variable $u \in \mathcal{X}_\kappa$ is a term. We define u's type to be $\tau(u) = \kappa$.
2. For all tuples $(f, \kappa_1,\ldots,\kappa_m \rightarrow \kappa, \kappa_{a_1}, \ldots, \kappa_{a_n}) \in F^T$, the arity of f is m and the symbol-arity of f is n. If $t_1, \ldots, t_m$ are terms and for all i with $1 \leq i \leq m$. $\tau(t_i) = \kappa_i$ then $f\langle\kappa_1,\ldots,\kappa_m \rightarrow \kappa, \kappa_{a_1}, \ldots, \kappa_{a_n}\rangle(t_1, \ldots, t_m)$ is also a term and its type $\tau(f\langle\kappa_1,\ldots,\kappa_m \rightarrow \kappa, \kappa_{a_1}, \ldots, \kappa_{a_n}\rangle(t_1, \ldots, t_m))$ is κ.

A term is *ground* if it contains no variables. We write u_κ to denote that the variable u is from the set $\mathcal{X}_\kappa$ and thus of a variable of the type symbol κ. Furthermore, we have no explicit quantifiers and all variables are implicitly universally quantified. Without loss of generality we also assume that all variables are named uniquely.

Definition 3.5.1.3 (Type-Symbol Literals). Let Σ^T be a signature. Then *literals* are defined as

1. $\bot$
2. $\top$
3. $s \approx t$ if $s,t \in T^T_\Sigma(\mathcal{X})$ and $\tau(s) = \tau(t)$
4. $s \not\approx t$ if $s,t \in T^T_\Sigma(\mathcal{X})$ and $\tau(s) = \tau(t)$

The equality predicate is a built-in overloaded (for each τ) predicate. We require both its left-hand and right-hand side terms to be of the same type, i.e. $\tau(s) = \tau(t)$. We call literals constructed by rules 1-3 *atoms*.

Definition 3.5.1.4 (Type-Symbol Clauses). We define *clauses* to be multisets of literals.

We use A for atoms, L for literals and C, D, ϕ for clauses. A clause is *ground* if it contains no variables.

Substitutions Here I introduce substitutions for type-symbol variables, terms and literals.

Definition 3.5.1.5 ((Variable Substitutions). A *substitution* is a mapping from variables ($\mathcal{X}$) to terms ($T^T_\Sigma(\mathcal{X})$).

We write them as $\sigma = \{u_1 \mapsto t_1, \ldots, u_n \mapsto t_n\}$, where all u_i are pairwise distinct and the type of the variable and terms are the same: $\tau(u_i) = \tau(t_i)$. We define them to be:

$$\sigma(v) = \begin{cases} t_i & \text{if } v = u_i \\ v & \text{otherwise} \end{cases}$$

We also write $u\sigma$ for $\sigma(u)$. A substitution σ is updated to return t for u (where $\tau(t) = \tau(u)$) by

$$\sigma[u \mapsto t](v) = \begin{cases} t & \text{if } v = u \\ \sigma(v) & \text{otherwise} \end{cases}$$

Definition 3.5.1.6 ((Term Substitutions). Substitutions are extended to non-variable terms by

$$f\langle \kappa_1,\ldots,\kappa_m \to \kappa,\ \kappa_{a_1},\ \ldots,\ \kappa_{a_n}\rangle(t_1,\ \ldots,\ t_m)\sigma \Rightarrow f\langle \kappa_1,\ldots,\kappa_m \to \kappa,\ \kappa_{a_1},\ \ldots,\ \kappa_{a_n}\rangle(t_1\sigma,\ \ldots,\ t_m\sigma)$$

Definition 3.5.1.7 ((Literal Substitutions). Substitutions are similarly extended to literals by

1. $\bot\sigma \Rightarrow \bot$
2. $\top\sigma \Rightarrow \top$
3. $(s \approx t)\sigma \Rightarrow s\sigma \approx t\sigma$
4. $(s \not\approx t)\sigma \Rightarrow s\sigma \not\approx t\sigma$

For clauses the substitution is applied to each literal of the clause and for clause sets to each clause of the set.

Unification Superposition requires unification and in particular the existence of a unique most general unifier (up to renaming). Therefore, I define unification of type-symbol terms and show that it computes a unique most general unifier.

Let t_1, t_2 be two terms, then unification is the task to derive a substitution σ such that $t_1\sigma = t_2\sigma$. The unification rules work on a tuple $(E;\ S)$, where E are the equations that are still to be unified and S are the equations that are already solved. We use $\doteq$ for not yet unified equations and $=$ for equations that are already solved. We start the unification of two terms t_1, t_2 with the tuple $(\{t_1 \doteq t_2\};\ \emptyset)$ and apply the following unification rules. If E is empty, the unification is complete and then S contains the unifier, since it only contains variable to term mappings. The unification has failed if any equation in E cannot be removed by applying one of the unification rules. In that case, no unification is possible. I define the rules in such a way that there is no overlap between them, this simplifies the proofs below. In particular, the side condition of the Decompose rule is only needed to prevent an overlap with the Identity rule.

$$\begin{array}{rrcl}
\textit{Identity} & (t \doteq t, E';\ S) & \Rightarrow & (E';\ S) \\
\textit{Decompose} & (f\langle\tau\rangle(t_1,\ldots,t_m) \doteq f\langle\tau\rangle(s_1,\ldots,s_m), E';\ S) & \Rightarrow & (t_1 \doteq s_1,\ldots,t_m \doteq s_m, E';\ S) \\
& & & \text{if } f\langle\tau\rangle(t_1,\ldots,t_m) \neq f\langle\tau\rangle(s_1,\ldots,s_m) \\
\textit{Orient} & (t \doteq u, E';\ S) & \Rightarrow & (u \doteq t, E';\ S) \\
& & & \text{if } u \text{ is a variable but } t \text{ is not} \\
\textit{Eliminate} & (u \doteq t, E';\ S) & \Rightarrow & (E'\{u \mapsto t\};\ u = t, (S\{u \mapsto t\})) \\
& & & \text{if } \tau(u) = \tau(t) \text{ and } u \notin vars(t)
\end{array}$$

In an actual implementation, the failure rules could also be considered, but since the type-symbol language is just an intermediate language, I am only interested in the existence of a unique most general unifier. Note in particular that the overall unification fails if we have an obligation of the form $f\langle\tau\rangle(\ldots) \doteq f\langle\tau'\rangle(\ldots)$ in E and $\tau \neq \tau'$.

Correctness and Uniqueness of Unification I now proof that this unification algorithm computes a unique most general unifier (up to renaming) and thus that such a unifier exists. To that end, I first show that S contains only variable to term mappings, i.e. that it actually contains a substitution. Then I show that the algorithm computes a unifier and always terminates. I continue by showing that no unifier is lost by any step of the algorithm and then conclude that it computes a most general unifier and that this most general unifier is unique.

Lemma 3.5.1.8 (S's Variables not in E)**.**
Let $(E;\ S) \Rightarrow (E';\ S')$ be one unification step, if E does not contain any variable that occurs in the left hand side of an equation of S then E' does not contain any variable that occurs in the left hand side of an equation of S'.

Proof. By induction over the unification rules. The Identity, Decompose and Orient rules do not change S and S' at all and do not change the variable occurring in E and E'. The Eliminate rule introduces a new left-hand side variable u in S' but instantiates all of u's occurrences in E'. It also instantiates u in the remainder of S', but by induction hypothesis u does not occur on the left hand side of the substitution represented by S. □

Now, I show that the computed mapping S is a substitution, because a unifier is a substitution that makes two terms syntactically equal.

Lemma 3.5.1.9 (S contains a substitution)**.**
Let t_1, t_2 be two terms, then $(t_1 \doteq t_2;\ \emptyset) \Rightarrow^* (E;\ S)$ is a series of unification steps then S only contains variable to term mappings.

Proof. By induction over the unification rules. The Identity, Decompose and Orient rule do not change S at all. The Eliminate rule introduces a new left hand side variable u to S' which cannot be instantiated, because it does not occur in E anymore (Lemma 3.5.1.8). □

Next, I show the algorithm is deterministic for each $\doteq$, i.e. for each unification obligation in E exactly one unification rule is applicable.

Lemma 3.5.1.10 (Type-Symbol Unification Computes a Unifier)**.**
If $(t \doteq s, E';\ S)$ and t unifiable with s, then exactly one rule of the unification algorithm is applicable on $t \doteq s$.

Proof. Suppose that $t = s$, then only the Identity rule is applicable.
Suppose that $t \neq s$ then we have four cases either t and s have a function symbol at the top, one of them is a variable and the other is not, or both of them are a variable:

- Let $t = f\langle\tau_t\rangle(t_1, \ldots, t_m)$ and $s = g\langle\tau_s\rangle(s_1, \ldots, s_m)$ both have function symbols at the top. Then they are only unifiable if the top function symbols ($f = g$) and their type definitions ($\tau_t = \tau_s$) are identical. In this case only the Decompose rule is applicable.
- Let $t = f\langle\tau_t\rangle(t_1, \ldots, t_m)$ have a function symbol at the top and $s = u$ be a variable. In this case only the Orient rule is applicable.
- Let $t = u$ be a variable and $s = g\langle\tau_s\rangle(s_1, \ldots, s_m)$ have a function symbol at the top. Then they are unifiable only if s does not contain t and they are of the same type. In this case only the Eliminate rule is applicable.
- Let $t = u_t$ and $s = u_s$ both be variables. In this case only the Eliminate rule is applicable, because $t \neq s$ and they are unifiable only it they are of the same type.

□

The unification algorithm must also terminate in order to be reasonable.

Lemma 3.5.1.11 (Type-Symbol Unification Terminates)**.**
Applying unification on $(\{t_1 \doteq t_2\};\ \emptyset)$ always terminates.

Proof. Applying unification decreases the lexicographic ordering on the pair of the number of distinct variables and the number of symbols occurring in E below a $\doteq$ obligation. This is not true for the Orient rule, but we can only apply it at most once for each $\doteq$ obligation. □

To show that the unification algorithm computes the most general unifier, I show that for each step of the unification algorithm no unifier is lost. Then, the most general unifier must be an instance (renaming) of the resulting substitution.

Lemma 3.5.1.12 (Type-Symbol Unification Preserves Unifiers)**.**
Let $(E;\ S) \Rightarrow (E',\ S')$ be one step of the unification algorithm, then if θ is a unifier for $(E,\ S)$ it is also a unifier for $(E',\ S')$.

Proof. Let $t \doteq s$ be an arbitrary unification obligation in E. Let E_r be E without $t \doteq s$. Suppose $t = s$, then the Identity rule is applicable and $E' = E_r$ and $S' = S$. Clearly, if θ is a unifier for $(E,\ S)$, then it is also a unifier for $(E',\ S')$. Suppose $t \neq s$, then it is one of the following cases:

- Let $t = f\langle\tau\rangle(t_1,\ldots,t_m)$ and $s = f\langle\tau\rangle(s_1,\ldots,s_m)$, then the Decompose rule is applicable and $E' = t_1 \doteq s_1,\ldots,t_m \doteq s_m,E_r$ and $S' = S$. Clearly, if θ is a unifier for $(E,\ S)$ then $t\theta = s\theta$ and thus it is also a unifier for the subterms $t_i\theta = s_i\theta$. It is therefore also a unifier for $(E',\ S')$.

- Let $t = f\langle\tau_t\rangle(t_1,\ldots,t_m)$ and $s = u$. Then the Orient rule uses commutativity of equality to reorient the unification obligation. Clearly, θ is still a unifier.

- Let $t = u$ and $s = g\langle\tau_s\rangle(s_1,\ldots,s_m)$. If θ is a unifier for $(E,\ S)$, then $t\theta = s\theta$. Therefore, θ remains a unifier if we replace t by s. After application of the Eliminate rule, E' is $E_r\{t \mapsto s\}$ and S' is $t = s, (S\{t \mapsto s\})$. Because we only instantiate t to s and move the uninstantiated obligation $t \doteq s$ to S', θ remains a unifier.

- Let $t = u_t$ and $s = u_s$. If θ is a unifier for $(E,\ S)$, then $t\theta = s\theta$. Therefore, θ remains a unifier if we replace t by s. After application of the Eliminate rule E' is $E_r\{t \mapsto s\}$ and S' is $t = s, (S\{t \mapsto s\})$. Because we only instantiate t to s and move the uninstantiated obligation $t \doteq s$ to S', θ remains a unifier.

□

Now I can combine the previous lemma to show that a most general unifier is computed and that it is unique up to renaming of the variables.

Lemma 3.5.1.13 (Type-Symbol Unification Computes a Most General Unifier)**.**
If there is a most general unifier θ of t and s, then $(\{t \doteq s\};\ \emptyset) \Rightarrow^* (\emptyset,\ S)$ and θ is an instance of the unifier contained in S.

Proof. Suppose there is a most general unifier and $(\emptyset,\ S)$ does not contain it. Since we know that the unification algorithm always terminates (Lemma 3.5.1.11) we have the following possibilities: Either (1.) $(\{t \doteq s\};\ \emptyset) \Rightarrow^* (E,\ S)$ and $E \neq \emptyset$ or (2.) $(\{t \doteq s\};\ \emptyset) \Rightarrow^* (\emptyset,\ S)$ and S does not contain the most general unifier. In the first case, Lemma 3.5.1.10 guarantees that there is a unification rule that is applicable if E is unifiable. E must be unifiable if there is a most general unifier, but then $(E,\ S)$ cannot be the end result of the unification algorithm. In the second case we know that no unifier (in particular the most general unifier) is lost by a step of the unification algorithm (Lemma 3.5.1.12). Therefore, the most general unifier is an instance of the unifier contained in S. □

Finally, I can conclude that there is a unique (up to renaming) most general unifier for any two type-symbol terms.

Lemma 3.5.1.14 (The Most General Unifier is Unique)**.**
The most general unifier of two type-symbol terms t and s is unique up to renaming of variables.

Proof. Follows from Lemma 3.5.1.13, because all most general unifiers are instances of the unifier σ contained in S. □

Semantics

Here, I define the semantics of the type-symbol language. In untyped first-order logic we have no concept of types or type symbols and their domains. In the type-symbol case we have type symbols which are part of the syntax and each type symbol is interpreted to represent a domain, a subset of the overall universe.

Definition 3.5.1.15 (Structure). A Σ^T-*structure* is a tuple $(\mathcal{U}^T, \mathcal{D}^T, \mathcal{I}^T_{\mathcal{T}}, \mathcal{I}^T_{\mathcal{F}})$, where

- $\mathcal{U}^T$ is a non-empty countable set of elements, the universe.
- $\mathcal{D}^T$ a non-empty set of non-empty disjoint subsets of $\mathcal{U}$. For each type symbol $\kappa \in S^T_{\mathcal{T}}$ there is a domain $D_\kappa \in \mathcal{D}^T$ that corresponds to it.
- $\mathcal{I}^T_{\mathcal{T}}$ is the set of interpretation function for the type symbols that maps them to their domains $(\mathcal{D}^T)$.
- $\mathcal{I}^T_{\mathcal{F}}$ is the set of interpretation function for the function symbols. For each function symbol $f \in S^T_{\mathcal{F}}$ with a declaration $(f,\ \kappa_1,\ldots,\kappa_m \to \kappa,\ \kappa_{a_1},\ \ldots,\ \kappa_{a_n}) \in \mathcal{F}^T$, there is a *function* $f^{\mathcal{I}}_{\kappa_1,\ldots,\kappa_m \to \kappa,\ \kappa_{a_1},\ \ldots,\ \kappa_{a_n}} \in \mathcal{I}^T_{\mathcal{F}}$ such that the function's argument domain is $\mathcal{I}^T(\kappa_1) \times \cdots \times \mathcal{I}^T(\kappa_m)$ and its range is $\mathcal{I}^T(\kappa)$.

The universe and functions are similar to untyped first-order logic. Unlike untyped first-order, the universe of the type-symbol language must be split into the domains the type symbols represent. Those subsets are represented by $\mathcal{D}^T$. Note that the set $\mathcal{D}^T$ is still countable, because there are only countable many non-empty disjoint subsets of a countable set (there are only countable many disjoint subsets of size one) and the universe $\mathcal{U}$ is countable. The type symbols must also be mapped to those domains by $\mathcal{I}^T_{\mathcal{T}}$.

Interpretation Here we define the interpretations of the type symbol language.

Definition 3.5.1.16 (Type Symbol Interpretation). The *interpretation* $(S^T_{\mathcal{T}} \to 2^{\mathcal{U}})$ of type symbols is defined by

$$\mathcal{I}^T(\kappa) = D_\kappa$$

where $D_\kappa \in \mathcal{D}$ represents the domain of elements of that type symbol.

A *variable valuation* $(\mathcal{V})$ is a mapping from $\mathcal{X} \to \mathcal{U}$ such that the result of the variable valuation $\mathcal{V}(u_\kappa)$ is always a member of the domain $\mathcal{I}_{\mathcal{V}}(\kappa)$ of the variable's type symbol κ. A variable valuation is always relative to a given Σ^T-structure, but we leave this relation implicit.

Definition 3.5.1.17 (Term Interpretation). The *interpretation* $(T_\Sigma(\emptyset) \to \mathcal{U})$ of terms is defined by

$$\begin{array}{lcl} \mathcal{I}^T_{\mathcal{V}}(u) & = & \mathcal{V}(u) \\ \mathcal{I}^T_{\mathcal{V}}(f\langle\tau\rangle(t_1,\ldots,t_m)) & = & f^{\mathcal{I}}_\tau(\mathcal{I}_{\mathcal{V}}(t_1),\ldots,\mathcal{I}_{\mathcal{V}}(t_m)) \end{array}$$

where $f^{\mathcal{I}}_\tau \in \mathcal{I}^T_{\mathcal{F}}$ is the function corresponding to the function symbol $f \in S^T_{\mathcal{F}}$ with type definition τ. All results of $f^{\mathcal{I}}_\tau$ must be elements of $\mathcal{I}^T_{\mathcal{T}}(\tau(f\langle\tau\rangle(t_1,\ldots,t_m)))$.

Definition 3.5.1.18 (Literal Interpretation). The *Interpretation* of literals is defined by

$$\begin{array}{lcl}
\mathcal{I}^T_{\mathcal{V}}(\bot) & = & 0 \\
\mathcal{I}^T_{\mathcal{V}}(\top) & = & 1 \\
\mathcal{I}^T_{\mathcal{V}}(s \approx t) & = & 1 \Leftrightarrow \mathcal{I}^T_{\mathcal{V}}(s) = \mathcal{I}^T_{\mathcal{V}}(t) \\
\mathcal{I}^T_{\mathcal{V}}(s \not\approx t) & = & 1 \Leftrightarrow \mathcal{I}^T_{\mathcal{V}}(s) \neq \mathcal{I}^T_{\mathcal{V}}(t)
\end{array}$$

The interpretation of a clause is true if the interpretation of any of its literals is true. The interpretation of a set of clauses is true if the interpretation for all the clauses the clause set contains is true. Let max be true if and only if at least one of its arguments is true, and min be true if and only if all of its arguments are true. Let $vars(\phi)$ be the set of all variables occurring in a clause ϕ. We extend interpretations to clauses by computing the set of variables and then universally quantifying over them.

Definition 3.5.1.19 (Clause Interpretation). The *interpretation* of clauses is defined by

$$\begin{array}{lcl}
\mathcal{I}_{\mathcal{V}}(\emptyset,\, \phi) & = & \max\limits_{L \in \phi}(\mathcal{I}_{\mathcal{V}}(L)) \\
\mathcal{I}_{\mathcal{V}}(\{u^{\kappa}\} \uplus V,\, \phi) & = & \min\limits_{e \in \mathcal{I}^T(\kappa)}(\mathcal{I}_{\mathcal{V}[u \mapsto e]}(V,\, \phi)) \\
\mathcal{I}(\phi) & = & \mathcal{I}_{\{\}}(vars(\phi),\, \phi))
\end{array}$$

Refutational Completeness - Ground Case

We know that untyped ground superposition is complete, therefore we relate ground type-symbol superposition to it. The differences between untyped and ground type-symbol superposition are the type-symbol declarations and type symbols. We now present a series of translations that translate ground type-symbol first-order logic to ground first-order logic and back. We then use them to prove the ground type-symbol superposition refutationally complete.

First, we translate pairs of type-symbol function symbols and their annotated type definitions (τ) to their untyped function symbol equivalent. The translation s maps each combination of function and its type definition to a new, fresh function symbol for the untyped ground case. By construction s is a bijection.

Then, the translation tv translates the terms between ground type-symbol and untyped. Using the translation s, the terms are transformed by mapping the type-symbol function and type symbol pairs to the single untyped function symbol:

$$tv(f\langle\tau\rangle(t_1,\, \ldots,\, t_m)) = s(f,\, \tau)(tv(t_1),\, \ldots,\, tv(t_m))$$

We denote the reverse translation by tv^{-1}:

$$tv^{-1}(s(f,\, \tau)(t_1,\, \ldots,\, t_m) = f\langle\tau\rangle(tv^{-1}(t_1),\, \ldots,\, tv^{-1}(t_m))$$

Well-typedness is preserved by superposition, because equations can only be between terms where the top type symbol is equal (Lemma 3.5.1.20). The translation $ts2ut$ translates type-symbol clauses and clause sets to untyped clauses and clause sets by translating all terms that occur in them. The untyped ordering $\prec'$ for the untyped terms and clauses is induced by the type-symbol ordering $\prec$ for the type-symbol terms and clauses before translation is used. The untyped ordering holds for $t \prec' s$ if and only if $tv^{-1}(t) \prec tv^{-1}(s)$. The selection of the type-symbol clauses is used for the translated untyped clauses.

We will now prove type-symbol and ground superposition to be equivalent by showing that, after translation, each inference in the one system is mirrored by an inference in the other system. We also need to show that $\prec'$ is a simplification ordering. To this end we first show that we can replace a subterm of a term t by a subterm of equal type without changing the type of the term t.

Lemma 3.5.1.20 (Type Judgement)**.**
For all terms $t[s]$ where s is t or a subterm of t it holds that $\tau(s) = \tau(s') \implies \tau(t[s]) = \tau(t[s'])$.

Proof. The proof is by induction over t.

- If t is s and is replaced by s' the lemma is trivial (i.e. if t is a variable or constant).
- If t is of the form $f\langle\kappa_1,\ldots,\kappa_m \to \kappa,\ \kappa_{a_1},\ \ldots,\ \kappa_{a_n}\rangle(t_1,\ \ldots,\ t_m)$ and one of its subterms t_i is replaced by t_i' and that subterm is still of $\tau(t_i) = \tau(t_i') = \kappa_i$ then by definition of terms (2) it holds that $\tau(t) = \kappa$.

□

We expect the inverse of the translation *tv* applied to the to result of an application of *tv* to result in the same term, thus the translation *tv* must be a bijection.

Lemma 3.5.1.21 (Bijection of *tv*)**.**
For all ground type-symbol terms t it holds that $tv^{-1}(tv(t)) = t$.

Proof. The proof is by induction over t.

- If t is a constant, then $f\langle\tau\rangle()$ is mapped to $s(f,\tau)()$ by *tv* which is mapped to $f\langle\tau\rangle()$ by tv^{-1}, which is the initial t.
- If t is not a constant, then it is of the form $f\langle\tau\rangle(t_1,\ \ldots,\ t_m)$. The mapping of the top level function and type symbol pair is preserved in the same way as it is for constants. The mapping of the arguments terms $t_1,\ \ldots,\ t_m$ results in the same terms by the induction hypothesis.

□

In order for $\prec'$ to be total, we need that terms that are not equal before the translation tv^{-1} are not equal afterwards.

Corollary 3.5.1.22 (Terms Stay Distinct)**.**
For all untyped ground terms t and s it holds that if $t \neq s$ then $tv^{-1}(t) \neq tv^{-1}(s)$.

Proof. Follows from Lemma 3.5.1.21. □

For the untyped superposition calculus to work, it needs a reduction ordering as a parameter. Therefore, I have to show that the translated ordering is still one. Since the ordering is in fact also a simplification ordering, I show that $\prec'$ is in fact a simplification ordering.

Lemma 3.5.1.23 ($\prec'$ is a Simplification Ordering)**.**
If $\prec$ is a simplification ordering for type-symbol first-order logic, the derived ordering $\prec'$ for untyped first-order logic is also a simplification ordering.

Proof. Suppose the lemma does not hold, then $\prec'$ is either not total, not irreflexive, not transitive or not compatible with Σ-operations or does not have the subterm property. We do not have to consider closedness under substitutions for the ground case. In the following, let s'_1, s'_2 and s'_3 be three untyped ground terms and let s_1 be $tv^{-1}(s'_1)$, s_2 be $tv^{-1}(s'_2)$ and s_3 be $tv^{-1}(s'_3)$.

1. If $\prec'$ is not total, then there must be an s'_1 and an s'_2 such that $s'_1 \neq s'_2$ and neither $s'_1 \prec' s'_2$ nor $s'_2 \prec' s'_1$. From Corollary 3.5.1.22 we know that $s_1 \neq s_2$. By applying the definition of $\prec'$, we get that neither $s_1 \prec s_2$ nor $s_2 \prec s_1$ can then hold. But then $\prec$ cannot be a simplification ordering.

2. If $\prec'$ is not irreflexive, then there must be an s'_1 such that $s'_1 \prec' s'_1$. By applying the definition of $\prec'$ we get that then there must also be an s_1 such that $s_1 \prec s_1$. But then $\prec$ cannot be a simplification ordering.

3. If $\prec'$ is not transitive, then there must be an s'_1, an s'_2 and s'_3 such that $s'_1 \prec' s'_2$ and $s'_2 \prec' s'_3$ but not $s'_1 \prec' s'_3$. By applying the definition of $\prec'$ we get that then there must also be an s_1, an s_2 and s_3 such that $s_1 \prec s_2$ and $s_2 \prec s_3$ but not $s_1 \prec s_3$. But then $\prec$ cannot be a simplification ordering.

4. If $\prec'$ is not compatible with Σ-operations, then there must be an s'_1 and an s'_2 such that $s'_1 \prec' s'_2$ does not imply $f(t_1,\ldots,s'_1,\ldots,t_m) \prec' f(t_1,\ldots,s'_2,\ldots,t_m)$. By applying the definition of $\prec'$ there must be an s_1 and s_2 such that $s_1 \prec s_2$ does not imply $f(t_1,\ldots,s_1,\ldots,t_m) \prec f(t_1,\ldots,s_2,\ldots,t_m)$. But then $\prec$ cannot be a simplification ordering.

5. Closedness under substitutions is not applicable since there are no substitutions in the ground case

6. If $\prec'$ does not have the subterm property, then there must be an s'_1 and an s'_2 such that s'_2 is a proper subterm of s'_1 but it does not hold that $s'_2 \prec' s'_1$. By applying the definition of $\prec'$ there must be an s_1 and s_2 such that it does not hold that $s_2 \prec s_1$. But then $\prec$ cannot be a simplification ordering.

□

To show that both the untyped and the type-symbol are bisimulations of each other, I show that each of their inferences can simulate the corresponding inference of the other calculus. I start with the Equality Resolution and the Equality Factoring inferences and then show the Negative Superposition and the Positive Superposition inferences.

Lemma 3.5.1.24 (Bisimulation Lemma for the Equality Resolution and Equality Factoring Inferences)**.** Let C be a ground type-symbol clause and $ts2ut(C)$ the translated clause in untyped ground first-order logic. Then there is an equality resolution (or equality factoring) inference of $ts2ut(C)$ if and only if there is an equality resolution (or equality factoring) inference from C.

Proof. Suppose there is a $ts2ut(C)$ so that there is an equality resolution inference for $ts2ut(C)$ such that the inference result is not a result of the equality resolution inference from C (or vice versa). Then there must be at least one side condition of the equality resolution inference that is fulfilled for $ts2ut(C)$ but not for C (or vice versa). The side conditions for the equality resolution inference (Sect. 3.3), simplified for the ground case by removing the unifier, are:

1. $s = s'$.
2. a) $(s \not\approx s')$ is maximal in $(C' \vee s \approx s')$ and no literal is selected or
   b) $s \not\approx s'$ is the selected literal.

We now show why each side condition must also be fulfilled for C if it is fulfilled for $ts2ut(C)$ and fulfilled for $ts2ut(C)$ if it is fulfilled for C.

1. In the ground case there are no unifiers. Since the translations are bijective, the equality is preserved by the translation.
2. a) $(s \not\approx s')$ is maximal in $(C' \vee s \approx s')$ if and only if it is maximal in the translated clause, because translated literals are compared by translating them back to type-symbol and comparing them there.
   b) $s \not\approx s'$ is the selected literal either in both or in neither because the same selection of literals is used for $ts2ut(C)$ and C.

Thus, if the equality resolution inference is possible for $ts2ut(C)$ if and only if it the inference is possible with C.

Suppose there is a ground $ts2ut(C)$ so that there is an equality factoring inference for $ts2ut(C)$ such that the inference result is not a result of the equality factoring inference from C. Then there must be at least one side condition of the equality factoring inference that is fulfilled for $ts2ut(C)$ but not for C. The side conditions for the equality factoring inference (Sect. 3.3), simplified by removing the unifier which do not occur in the ground case, are:

1. $s = t$
2. $t \approx t'$ maximal in $C' \vee s \approx s' \vee t \approx t'$ and no literal is selected.
3. $s' \not\succeq s'$ and $t \not\succeq t'$

We now show why each side condition must also be fulfilled for C if it is fulfilled for $ts2ut(C)$ (and vice versa).

1. As before in the ground case, there are no unifiers. Since the translations are bijective the equality is preserved by the translation.
2. The orderings and selection are the same, because the ordering of C is used for $ts2ut(C)$.
3. The orderings are the same, because the ordering of C is used for $ts2ut(C)$.

Thus if the equality factoring inference is possible for $ts2ut(C)$, it is also possible for the type-symbol inference with C. □

Lemma 3.5.1.25 (Bisimulation Lemma for the Superposition Inferences)**.**
Let $C = C' \vee s \approx s'$ and $D = D' \vee t \approx t'$ be two ground type-symbol clauses. There is a superposition inference between $ts2ut(C)$ and $ts2ut(D)$ where $ts2ut(s)$ and some subterm of $ts2ut(t)$ if and only if there is a superposition inference between C and D.

Proof. Suppose there is a $ts2ut(C)$ and $ts2ut(D)$ so that there is a positive superposition inference for $ts2ut(C)$ and $ts2ut(D)$ such that the inference result is not a result of the positive superposition inference from C and D (or vice versa). Then there must be at least one side condition of the positive superposition inference that is fulfilled for $ts2ut(C)$ and $ts2ut(D)$ but not for C and D (or vice versa). The side conditions for the positive superposition inference (Sect. 3.3), simplified by removing the unifier, are:

1. $s \npreceq s'$ and $t \npreceq t'$
2. Not applicable (There are no variables in the ground case.)
3. $t = s_2$
4. $(t \approx t')$ is strictly maximal in $(D' \vee t \approx t')$ and no literal is selected
5. $(s \approx s')$ is strictly maximal in $(C' \vee s \approx s')$ and no literal is selected

We now show why each side condition must also be fulfilled for C and D if it is fulfilled for $ts2ut(C)$ and $ts2ut(D)$ (and vice versa).

1. $s \npreceq s'$ and $t \npreceq t'$ hold for both, because the orderings of the translated and original terms is the same.
2. -
3. The translation preserves all equalities.
4. The orderings and selection are the same, because the ordering of D is used for $ts2ut(D)$.
5. The orderings and selection are the same, because the ordering of C is used for $ts2ut(C)$.

Thus, the positive superposition inference is possible for $C\theta$ and $D\theta$ if and only if it is also possible for the type-symbol inference with C and D.

Suppose there is a ground $ts2ut(C)$ and $ts2ut(D)$ so that there is a negative superposition inference for $ts2ut(C)$ and $ts2ut(D)$ such that the inference result is not a result of the negative superposition inference from C and D (or vice versa). Then there must be at least one side condition of the negative superposition inference that is fulfilled for $ts2ut(C)$ and $ts2ut(D)$ but not for C and D (or vice versa). The side conditions for the negative superposition inference (Sect. 3.3), simplified by removing the unifier, are:

1. $s \npreceq s'$ and $t \npreceq t'$
2. Not applicable (There are no variables in the ground case.)
3. $t = s_2$

4. $t \approx t'$ is strictly maximal in $D' \vee t \approx t'$ and no literal is selected

5. a) $s \not\approx s'$ is maximal in $C' \vee s \not\approx s'$ and no literal is selected or
   b) $s \not\approx s'$ is selected

We now show why each side condition must also be fulfilled for C and D if it is fulfilled for $ts2ut(C)$ and $ts2ut(D)$ (and vice versa). The side conditions and proofs 1. to 4. are identical to the side conditions and proofs of positive superposition inference.

5. a) $s \not\approx s'$ is maximal in both since the same ordering is used for translated and original literals.
   b) If $s \not\approx s'$ is selected in $ts2ut(C)$ it is also selected in C, since the selection is the same for both clauses.

Thus, the negative superposition inference is possible for $ts2ut(C)$ and $ts2ut(D)$ if and only if it is also possible for the non-ground inference with C and D. □

We have now shown that after translation type-symbol and ground superposition are bisimulating each other. It remains to show that the clause sets before and after translation are equisatisfiable.

Lemma 3.5.1.26 (Type-Symbol Satisfiable Implies Untyped Satisfiable)**.**
Let $(\mathcal{U}^T, \mathcal{I}_F^T, \mathcal{D})$ be a ground type-symbol structure, then there is an untyped structure $(\mathcal{U}, \mathcal{I}_F)$ such that for all type-symbol ground terms t it holds that $\mathcal{I}^T(t) = \mathcal{I}(tv(t))$.

Proof. Let $\mathcal{U} = \mathcal{U}^T$ and $\mathcal{I}_F = \mathcal{I}_F^T$ and let for each untyped function symbol $f_{s(f^T,\kappa)}$ the corresponding function be $f_\kappa^{\mathcal{I}}$. Then the remaining proof is by induction over t. If t is a constant $f^T\langle\kappa\rangle()$, then its translation is $s(f^T,\kappa)()$ and both their interpretations are $f_\kappa^{\mathcal{I}}()$. If t is not a constant, then it is of the form $f^T\langle\tau\rangle(t_1,\ldots,t_m)$ and its translation is $s(f^T,\tau)(tv(t_1),\ldots,tv(t_m))$. The interpretation of t is $f_\tau^{\mathcal{I}}(\mathcal{I}^T(t_1),\ldots,\mathcal{I}^T(t_m))$ and of its translation is $f_\tau^{\mathcal{I}}(\mathcal{I}(tv(t_1)),\ldots,\mathcal{I}(tv(t_1)))$. From the induction hypothesis it follows that for all i with $1 \leq i \leq m$ it holds that $\mathcal{I}(t_i) = \mathcal{I}(tv(t_i))$. Thus, their interpretations are equal. □

Lemma 3.5.1.27 (Untyped and Type-Symbol Interpretations are Equisatisfiable)**.**
Let N be a clause set of ground type-symbol first-order logic it is satisfiable if and only if the translated clause set N' in untyped first-order logic is satisfiable.

Proof. $\Rightarrow$ The proof is by induction over the interpretation rules from Section 3.5.1 and Section 2.1.2. The interpretation rules for ground clause sets, clauses and literals are identical. From Lemma 3.5.1.26 we know that there is a untyped structure so that the interpretation of the translated term is the same as the original interpretation.

$\Leftarrow$ The direction from an untyped to a type-symbol model is slightly more complicated. This is because in the untyped setting there is no separation of the domains $\mathcal{D}$ of the type-symbol setting. Let $(\mathcal{U},\mathcal{I}_F)$ be the untyped structure. For each type symbol κ in S_T^T all elements $e \in \mathcal{U}$ become new elements e^κ of the domain of κ, i.e. $e^\kappa \in \mathcal{I}^T(\kappa)$. The new universe $\mathcal{U}^T$ then is the union of all domains and $\mathcal{D}^T$ is the set of all domains. The functions $f_\tau \in \mathcal{I}_F^T$ are constructed by combining $s(f,\tau) \in \mathcal{I}_F$ and $(f,\kappa,\kappa_1,\ldots,\kappa_m) \in \mathcal{F}^T$ such that $f^{\mathcal{I}}_\tau(e_1^{\kappa_1},\ldots,e_m^{\kappa_m}) = e^\kappa$ if and only if $s(f,\tau)^{\mathcal{I}}(e_1,\ldots,e_m) = e$. Where $e,e_1,\ldots,e_m \in \mathcal{U}$ and $e^\kappa, e_1^{\kappa_1},\ldots,e_m^{\kappa_m}$ are from the respective $\mathcal{I}^T(\kappa_i)$.

Again, the proof is by induction over the interpretation rules and again the interpretation rules for ground clause sets, clauses and literals are identical. It is therefore necessary to show that $\mathcal{I}(t) = \mathcal{I}(s)$ implies $\mathcal{I}^T(tv^{-1}(t)) = \mathcal{I}^T(tv^{-1}(s))$ for all untyped ground terms t and s. We know that type-symbol equations must be of the same type-symbol κ on the left hand and right hand side, which guarantees that their interpretations both are elements of $\mathcal{I}^T(\kappa)$. We have constructed the type-symbol interpretation by duplicating each untyped element for each type-symbol κ. Let iso be a mapping that maps for all κ all e^κ back to the e of the untyped model. Then $\tau(tv^{-1}(t)) = \tau(tv^{-1}(s))$, $\mathcal{I}(t) = \mathrm{iso}(\mathcal{I}^T(tv^{-1}(t)))$ and $\mathcal{I}(s) = \mathrm{iso}(\mathcal{I}^T(tv^{-1}(s)))$ imply that $\mathcal{I}(t) = \mathcal{I}(s)$ implies $\mathcal{I}^T(tv^{-1}(t)) = \mathcal{I}^T(tv^{-1}(s))$.

It remains to show that for all untyped ground terms t it holds that $\mathcal{I}(t) = \mathrm{iso}(\mathcal{I}^T(tv^{-1}(t)))$. The proof is by induction over t. If t is a constant $s(f^T,\kappa)()$ then its translation to ground type-symbol is $f^T\langle\kappa\rangle()$ and if $\mathcal{I}(s(f^T,\kappa)()) = e$ then $\mathcal{I}^T(f^T\langle\kappa\rangle()) = e^\kappa$ by construction of our type-symbol model. Clearly $e = \mathrm{iso}(e^\kappa)$. If t is not a constant, then it is of the form $s(f^T,\kappa_1,\ldots,\kappa_m \to \kappa)(t_1,\ldots,t_m)$ and its translation is $f^T\langle\kappa_1,\ldots,\kappa_m \to \kappa\rangle(tv^{-1}(t_1),\ldots,tv^{-1}(t_m))$. The untyped interpretation is $f^{\mathcal{I}}(\mathcal{I}(t_1),\ldots,\mathcal{I}(t_m)) = e$ and while by induction hypothesis $f^{\mathcal{I}}_{\kappa_1,\ldots,\kappa_m\to\kappa}(\mathcal{I}^T(tv^{-1}(t_1)), \ldots, \mathcal{I}^T(tv^{-1}(t_m)))$ is the type-symbol one. $\mathcal{I}(t_i) = \mathrm{iso}(\mathcal{I}^T(tv^{-1}(t_i)))$ for all argument terms. Thus, by construction of $f^{\mathcal{I}}_{\kappa_1,\ldots,\kappa_m\to\kappa}$ the type-symbol interpretation is $f^{\mathcal{I}}_{\kappa_1,\ldots,\kappa_m\to\kappa}(\mathcal{I}^T(tv^{-1}(t_1)),\ldots, \mathcal{I}^T(tv^{-1}(t_m))) = e^\kappa$. It follows that $e = \mathrm{iso}(e^\kappa)$. $\square$

Since I showed the translation to be satisfiability preserving and I showed the type-symbol superposition calculus and the translated untyped one to be bisimulations of each other I have shown completeness for ground type-symbol superposition. As the ground type-symbol language is an intermediate language in the overall proof, I will not show dynamic refutational completeness for it.

Theorem 3.5.1.28 (Static Refutational Completeness)**.**
Superposition is refutationally complete for the ground version of type-symbol first-order logic.

Proof. We have shown that our translations preserve satisfiability (Lemma 3.5.1.27) and we have shown that the ground type-symbol superposition calculus and the translated ground untyped superposition calculus are bisimulations of each other (Lemma 3.5.1.24 and Lemma 3.5.1.25). Therefore the completeness of the ground untyped version of superposition (Lemma 2.2.4.11) is lifted to the ground type-symbol version of superposition. $\square$

3.5.2. Lifting to Type Symbols with Term Variables

The syntax and semantics for the type-symbol language with variables are already given in the previous section (Sect. 3.5.1). There, completeness was shown for the ground type-symbol case.

Refutational Completeness

I will now lift ground type-symbol first-order logic, proved refutationally complete in the previous section, to non-ground type-symbol first-order logic. In this lifting step the encoding is the replacement of a non-ground clause by a set of ground clauses, just as in the lifting in the untyped proof.

As a first step to justify the restriction of superposition to not superpose into variables, I have to show that variables are closed under rewriting.

Lemma 3.5.2.1 (Variables are Closed Under Rewriting)**.**
Let $C \in N$ and θ be a substitution such that $C\theta \in G_\Sigma(C)$ and x be a variable occurring in C then: If $x\theta \to t$ is some rewrite step, then there exists a θ' such that $x\theta' = t$ and $y\theta' = y\theta$ for every variable $y \neq x$ and furthermore $C\theta' \in G_\Sigma(C)$.

Proof. From the definition of literals it follows that only terms of the same type symbol can be equal. Rewriting is only possible by replacing terms by other terms of the same type symbol, which does not change the overall type of a term (Lemma 3.5.1.20). Since a variable can be instantiated to any term of the type symbol it is assigned to, the lemma holds. □

To lift the completeness from the ground type-symbol case to the type-symbol case with variables, the inferences need to be lifted. I first lift the Equality Resolution and the Equality Factoring inferences and then the Positive Superposition and the Negative Superposition inferences into non-variable positions.

Lemma 3.5.2.2 (Lifting Lemma for the Equality Resolution and Equality Factoring Inferences)**.**
Let C be a clause and let θ be a substitution such that $C\theta$ is ground type-symbol first-order logic. Then every equality resolution (or equality factoring) inference of $C\theta$ is an instance of an equality resolution (or equality factoring) inference from C.

Proof. Suppose there is a ground $C\theta$ so that there is an equality resolution inference for $C\theta$ such that the inference result is not a result of the equality resolution inference from C. Then there must be at least one side condition of the equality resolution inference that is fulfilled for $C\theta$ but not for C. The side conditions for the equality resolution inference (Sect. 3.3) are:

1. $s\sigma = s'\sigma$, where σ is the most general unifier of s and s'.
2. a) $(s \not\approx s')\sigma$ *maximal* in $(C' \vee s \not\approx s')\sigma$ and no literal is selected or
   b) $s \not\approx s'$ is the selected literal.

I now show why each side condition must also be fulfilled for C if it is fulfilled for $C\theta$.

1. If the ground inference is applicable for $C\theta$, then $s\theta$ must be equal to $s'\theta$. By the existence and uniqueness of the most general unifier, there must be a most general unifier θ_g of s and s' such that $\theta = \theta_g\theta'$. But then this side condition is also fulfilled for C.
2. a) If $(s \approx s')\theta$ is maximal in $(C' \vee s \approx s')\theta$ and nothing is selected then there is no literal L in C' such that $L\theta \succ (s \approx s')\theta$. A literal is maximal with respect to a set of literals if there is a ground instance that is not smaller than any other literals in that instantiated set. Therefore, the uninstantiated literal is also maximal.
   b) If $s \not\approx s'$ is selected in $C\theta$ it is also selected in C.

Thus, if the equality resolution inference is possible for $C\theta$ it is also possible for the non-ground inference with C.

Suppose there is a ground $C\theta$ so that there is an equality factoring inference for $C\theta$ such that the inference result is not a result of the equality factoring inference from C. Then there must be at least one side condition of the equality factoring inference that is fulfilled for $C\theta$ but not for C. The side conditions for the equality factoring inference (Sect. 3.3) are:

1. $s\sigma = t\sigma$, where σ is the most general unifier of s and t.
2. $(t \approx t')\sigma$ maximal in $(C' \vee s \approx s' \vee t \approx t')\sigma$ and no literal is selected.
3. $s\sigma \not\preceq s'\sigma$ and $t\sigma \not\preceq t'\sigma$

I now show why each side condition must also be fulfilled for C if it is fulfilled for $C\theta$.

1. If the ground inference is applicable for $C\theta$, then $s\theta$ must be equal to $t\theta$. By the existence and uniqueness of the most general unifier, there must be a most general unifier θ_g of s and t such that $\theta = \theta_g\theta'$. But then this side condition is also be fulfilled for C.
2. Since $(t \approx t')\theta$ is maximal in $(C' \vee s \approx s' \vee t \approx t')\theta$ and no literal is selected, there is no literal L in C' such that $L\theta \succ (t \approx t')\theta$. Therefore, the uninstantiated literal is also maximal.
3. Since the ordering is total on ground terms, I know that $s\theta \succ s'\theta$ and $t\theta \succ t'\theta$. The ordering is also closed under substitutions and thus $s \not\preceq s'$ and $t \not\preceq t'$.

Thus, if the equality factoring inference is possible for $C\theta$ it is also possible for the non-ground inference with C. □

Lemma 3.5.2.3 (Lifting Lemma for the Superposition Inferences)**.**
Let $C = C' \vee s \approx s'$ and $D = D' \vee t \approx t'$ be two clauses without common variables and let θ be a substitution such that $C\theta$ and $D\theta$ are ground type-symbol first-order logic. If there is a superposition inference between $C\theta$ and $D\theta$ where $s\theta$ and some subterm of $t\theta$ are overlapped and $s\theta$ does not occur in $t\theta$ at or below a variable position of t, then the inference is an instance of a superposition inference from C and D.

Proof. Suppose there is a ground $C\theta$ and $D\theta$ so that there is a positive superposition inference for $C\theta$ and $D\theta$ such that the inference result is not a result of the positive superposition inference from C and D. Then there must be at least one side condition of the positive superposition inference that is fulfilled for $C\theta$ and $D\theta$ but not for C and D. The side conditions for the positive superposition inference (Sect. 3.3) are:

1. $s\sigma \not\preceq s'\sigma$ and $t\sigma \not\preceq t'\sigma$
2. s_2 is not a term variable
3. $t\sigma = s_2\sigma$ and σ is the most general unifier of s and s_2
4. $(t \approx t')\sigma$ is strictly maximal in $(D' \vee t \approx t')\sigma$ and no literal is selected

5. $(s \approx s')\sigma$ is strictly maximal in $(C' \vee s \approx s')\sigma$ and no literal is selected

I now show why each side condition must also be fulfilled for C and D if it is fulfilled for $C\theta$ and $D\theta$.

1. Since the ordering is total on ground terms, I know that $s\theta \succ s'\theta$ and $t\theta \succ t'\theta$. The ordering is also closed under substitutions and thus $s \not\preceq s'$ and $t \not\preceq t'$.
2. That the inference is not at or below a variable position is part of the assumptions of the lemma.
3. If the ground inference is applicable for $C\theta$, then $t\theta$ must be equal to $s_2\theta$. By the existence and uniqueness of the most general unifier, there must be a most general unifier θ_g of t and s_2 such that $\theta = \theta_g\theta'$. But then this side condition is also be fulfilled for C and D.
4. If $(t \approx t')\theta$ is strictly maximal in $(D' \vee t \approx t')\theta$ and no literal is selected then there is no literal L in D' such that $L\theta \succeq (t \approx t')\theta$. Therefore, the uninstantiated literal is also maximal.

Thus if the positive superposition inference is possible for $C\theta$ and $D\theta$ it is also possible for the non-ground inference with C and D.

Suppose there is a ground $C\theta$ and $D\theta$ so that there is a negative superposition inference for $C\theta$ and $D\theta$ such that the inference result is not a result of the negative superposition inference from C and D. Then there must be at least one side condition of the negative superposition inference that is fulfilled for $C\theta$ and $D\theta$ but not for C and D. The side conditions for the negative superposition inference (Sect. 3.3) are:

1. $s\sigma \not\preceq s'\sigma$ and $t\sigma \not\preceq t'\sigma$
2. s_2 is not a term variable
3. $t\sigma = s_2\sigma$ and σ is the most general unifier of s and s_2
4. $(t \approx t')\sigma$ is strictly maximal in $(D' \vee t \approx t')\sigma$ and no literal is selected
5. a) $(s \not\approx s')\sigma$ is maximal in $(C' \vee s \approx s')\sigma$ and no literal is selected or
   b) $s \not\approx s'$ is selected

I now show why each side condition must also be fulfilled for C and D if it is fulfilled for $C\theta$ and $D\theta$. The side conditions and proofs 1. to 4. are identical to the side conditions and proofs of positive superposition inference.

5. a) If $(s \approx s')\theta$ maximal in $(C' \vee s' \approx s')\theta$ and nothing is selected then there is no literal L in C' such that $L\theta \succ (s \approx t)\theta$. Therefore, there is also no L in D such that $L \succ (t \approx t')$, otherwise the ordering would not be closed under substitutions.
   b) If $s \not\approx s'$ is selected in $C\theta$ it is also selected in C.

Thus, if the negative superposition inference is possible for $C\theta$ and $D\theta$ it is also possible for the non-ground inference with C and D. □

The lifting lemma for superposition only holds for superposition into non-variable position. Therefore, I also have to show that superposition into the variable positions is irrelevant.

Lemma 3.5.2.4 (Lifting Lemma for the Superposition Inferences Into Variables)**.**
Let $D = D' \vee t \approx t'$ and $C = C' \vee s \approx s'$ be two clauses without common variables and let θ be a substitution such that $C\theta$ and $D\theta$ are ground clauses of type-symbol first-order logic. If there is a superposition inference from $D'\theta \vee t\theta \approx t'\theta$ into $C'\theta \vee (s[s_2]_p)\theta \approx s'\theta$ (i.e. such that $t\theta = s_2\theta$ is rewritten to $t'\theta$) and $(s_2\theta)$ occurs at or below a variable position in s, then the inference result is identical to a ground instance of D or redundant.

Proof. Let u be the variable of C under which $s_2\theta$ occurs, i.e. $s_2\theta$ is a subterm of $u\theta$, i.e. $u\theta[s_2\theta]_{p'}$. Since variables are closed under rewriting (Lemma 3.5.2.1), u can also be instantiated to take the rewriting of $s_2\theta$ to $t'\theta$ into account. Let θ' be the corresponding substitution, i.e. $\theta' = \theta[u \mapsto (u\theta[t'\theta]_{p'})]$.

The resulting clause of the superposition inference is $R = D'\theta \vee C'\theta \vee (s[t']_p)\theta \approx s'\theta$. If D' is empty and u occurs only once in C, then the resulting clause R is $C\theta'$ and thus it is identical to a ground instance of D. If D' is not empty or u occurs more than once in C, then clearly each occurrence of $u\theta$ in R can be rewritten to $u\theta'$ by conditional rewriting with $D\theta$. The result of this rewrite step is either a ground instance $C\theta'$ of C (if D' is empty) or is redundant with respect to that ground instance $C\theta'$. □

I now show that the fact that variables are closed under rewriting and that the lifting lemmas hold which is sufficient for lifting a proof of the ground type-symbol case to the non-ground type-symbol case.

Lemma 3.5.2.5 (Lifting)**.**
Let N be a set of type-symbol clauses. Then $\bot \in \text{Sup}^*(G_\Sigma(N))$ implies $\bot \in \text{Sup}^*(N)$.

Proof. Each ground Equality Resolution inference and each ground Equality Factoring inference can be replayed in the non-ground case (Lemma 3.5.2.2). Each ground Superposition inference, that is not into a variable position, can be replayed in the non-ground case (Lemma 3.5.2.3). Furthermore, each result of a ground Superposition inference, into a variable position, is redundant (Lemma 3.5.2.4). Because of these lifting lemmas (Lemma 3.5.2.2, 3.5.2.3 and 3.5.2.4), each step of the ground type-symbol proof (in $\text{Sup}^*(G_\Sigma(N))$) is mirrored in the type-symbol clause set ($\text{Sup}^*(N)$). □

Furthermore, I prove that any model of the ground type-symbol case can be lifted to a model of the non-ground type-symbol case.

Lemma 3.5.2.6 (Model Lifting)**.**
Let N be a countable set of type-symbol clauses and let $\mathcal{I}$ be an interpretation. Then $\mathcal{I} \models G_\Sigma(N)$ implies that there is an interpretation $\mathcal{I}'$ such that $\mathcal{I}' \models N$.

Proof. Let the structure $(\mathcal{U}^T, \mathcal{D}^T, \mathcal{I}^T_\mathcal{T}, \mathcal{I}^T_\mathcal{F})$ be the structure of the interpretation of $G_\Sigma(N)$. Let $\mathcal{D}^T_v$ be $\mathcal{D}^T$ such that for each $d \in \mathcal{D}^T$ and each $e \in d$ there is a ground term t in $G_\Sigma(N)$ such that e is the interpretation of t. For each $d \in \mathcal{D}^T$ and let d_v be the union of all $e \in d$ such that there is a ground term t in $G_\Sigma(N)$ such that e is the interpretation of t. Let $\mathcal{D}^T_v$ be the union of all such

d_v and let $\mathcal{U}_v^T$ be the union of all such d_v. Clearly, if the ds are pair-wise disjoint, then the d_vs are also pair-wise disjoint. It also holds that $\mathcal{I}' \vDash G_\Sigma(N)$ with the structure $(\mathcal{U}_v^T, \mathcal{D}_v^T, \mathcal{I}_\mathcal{T}^T, \mathcal{I}_\mathcal{F}^T)$, because we only removed elements of the universe that are not used in the interpretation. When using the structure $(\mathcal{U}_v^T, \mathcal{D}_v^T, \mathcal{I}_\mathcal{T}^T, \mathcal{I}_\mathcal{F}^T)$, there is no difference between the interpretation of a type-symbol κ and the union of the interpretations of all ground instances of that type, because all elements that do not have a corresponding ground term were removed. Therefore, $\mathcal{I} \vDash G_\Sigma(N)$ implies $\mathcal{I}' \vDash N$. □

I also show that a set of type-symbol clauses is satisfiable if and only if the corresponding ground type-symbol clauses are satisfiable.

Lemma 3.5.2.7 (Type-Symbol and Ground Type-Symbol are Equisatisfiable)**.**
A countable set N of Σ-clauses is satisfiable if and only if its monomorphic version $G_\Sigma(N)$ is satisfiable.

Proof. By case distinction whether $G_\Sigma(N)$ is satisfiable.

- Suppose $G_\Sigma(N)$ does not have a model, then $\bot \in \mathrm{Sup}^*(G_\Sigma(N))$. By lifting (Lemma 3.5.2.5) we have $\bot \in \mathrm{Sup}^*(N)$ and thus N does not have a model.
- Suppose $G_\Sigma(N)$ has a model, then there is an interpretation $\mathcal{I}$ such that $\mathcal{I} \vDash G_\Sigma(N)$. By model lifting (Lemma 3.5.2.6) we also have $\mathcal{I} \vDash N$.

□

Lifting the proofs and models from the ground type-symbol case to the non-ground one is sufficient to show that refutational completeness of superposition for the type-symbol language follows. Again, because the type-symbol language is an intermediate language in the overall proof, I will not show dynamic refutational completeness for it.

Theorem 3.5.2.8 (Static Refutational Completeness)**.**
Superposition is refutationally complete for type-symbol first-order logic.

Proof. I have shown that a countable set N of type-symbol clauses is satisfiable if and only if its ground version $G_\Sigma(N)$ is satisfiable (Lemma 3.5.2.7). Furthermore, I have shown that a proof in the monomorphic version is lifted to the polymorphic version (Lemma 3.5.2.5). Therefore, the completeness of the ground type-symbol version of superposition (Lemma 3.5.1.28) is lifted to the type-symbol version of superposition. □

3.5.3. Lifting to Monomorphism

Here, I proof refutational completeness of a monomorphic first-order language. Like the type-symbol language, the *monomorphic* language allows terms with variables. But now instead of type symbols, I allow complex type terms like in the polymorphic language with type classes, but without the type variables, type classes and type-class constraints. I will reuse the polymorphic language with type classes (Section 3.1), but restrict the syntax to disallow type variables in terms and clauses. By removing the type variables, I also remove the type classes (and constraints). As in the previous two completeness proofs, I will omit predicates and only consider clauses instead of formulas.

Syntax

I reuse the polymorphic signature and syntax (Section 3.1.1), but disallow the use of type variables in the type terms occurring in clauses and formulas. This is done by removing the first rule in the definition of type terms (Def. 3.1.1.3). Thus, type variables can only be used in the declarations.

Signature Even though I reuse the original signature (Def. 3.1.1.1), only a part of it is necessary. After well-typedness of the initial clause set is established, the unnecessary part of the signature can be removed. The relevant part of the signature for the monomorphic first-order language without type variables is a tuple $\Sigma^M = (S_{\mathcal{F}}, S_{\mathcal{T}}, \mathcal{F})$ where $S_{\mathcal{F}}$ is the set of function symbols, $S_{\mathcal{T}}$ is the set of type constructor symbols and $\mathcal{F}$ declares the arity and the argument and return types.

The other elements are not required, after the initial well-typedness check for the terms that occur in clauses. The predicate symbols ($S_{\mathcal{P}}$) and predicate type declarations ($\mathcal{P}$) are not necessary, since I omit predicates for the completeness proof. The type class symbols ($S_{\mathcal{K}}$) and subclass class declarations ($\mathcal{TC}$) are not necessary, since I disallow type variables and thus type classes. The type declarations ($\mathcal{T}$) are not necessary, since the clause set only contains ground types and does not introduce new ground type terms.

Semantics

I reuse the polymorphic semantics without changes (Sect. 3.1.4). The definition of a structure is given in Definition 3.1.4.1. Due to the restricted syntax, the rules concerning type variables (first rule of Def. 3.1.4.2), type classes (Def. 3.1.4.5) and type-class constraints (Def. 3.1.4.4) cannot apply.

Refutational Completeness

I have already shown that type-symbol superposition is refutationally complete (Theorem 3.5.2.8), therefore I translate the monomorphic language to it for the refutational completeness proof. The proof is similar to the lifting from ground untyped first-order logic to ground type-symbol first-order logic. The main differences between the type-symbol language and the monomorphic language is, that the type-symbol one only has type symbols whereas the monomorphic one has complex type terms (both have no type variables).

I now present the translations between monomorphic and type-symbol first-order language. The translation *ty* maps pairs of function symbols and a list of ground type terms to distinct type symbols, i.e. $ty(f, \tau_1, \ldots, \tau_n) = \kappa$ such that $ty(f, \tau_1, \ldots, \tau_n) = ty(f_r, \tau_{r_1}, \ldots, \tau_{r_n})$ if and only if $f = f_r$ and for all i it holds that $\tau_i = \tau_{r_i}$. The translation *ty* is bijective by construction.

First, I need to introduce notation to refer to the type of a function term and its translation. I define $\tau(f, \tau_r)$ to be $ty(\tau_r)$. Let furthermore κ be a type symbol and σ' be such that $\tau = \tau^f \sigma'$ then I define $\tau^{-1}(f, \kappa)$ to be $\alpha_1\sigma', \ldots, \alpha_n\sigma'$.

The translation *tv* translates the monomorphic terms to type symbol terms. For the translation I annotate the term variables in the subscript with their corresponding type term respectively type symbol.

$$\begin{aligned} tv(u_\tau) &\Rightarrow u_{ty(\tau)} \\ tv(f\langle\tau_r,\,\tau_1,\,\ldots,\,\tau_n\rangle(t_1,\,\ldots,\,t_m)) &\Rightarrow f\langle ty',\,ty(\tau_1),\,\ldots,\,ty(\tau_n)\rangle(tv(t_1),\,\ldots,\,tv(t_m)) \end{aligned}$$

where ty' is defined as

$$ty' = ty(\tau(t_1)),\,\ldots,\,ty(\tau(t_m)) \to ty(\tau_r)$$

Then the translation tv^{-1} is the reverse translation.

$$\begin{aligned} tv^{-1}(u_\kappa) &\Rightarrow u_{ty^{-1}(\kappa)} \\ tv^{-1}(f\langle\kappa_1,\ldots,\kappa_m \to \kappa,\,\kappa_{a_1},\,\ldots,\,\kappa_{a_n}\rangle(t_1,\,\ldots,\,t_m)) &\Rightarrow f\langle\tau'\rangle(tv^{-1}(t_1),\,\ldots,\,tv^{-1}(t_m)) \end{aligned}$$

where τ' is defined as

$$\tau' = ty^{-1}(\kappa),\,ty^{-1}(\kappa_{a_1}),\,\ldots,\,ty^{-1}(\kappa_{a_n})$$

For the ordering $\prec'$ of the translated terms and clauses, the ordering $\prec$ of the terms and clauses before translation is used. Formally: $t \prec' s$ if and only if $tv^{-1}(t) \prec tv^{-1}(s)$. Finally, the translation *m2ts* translates monomorphic clauses to type-symbol clauses by translating all monomorphic terms in them to type-symbol terms. The selection of the clauses is used for the translated clauses.

The type-symbol signature uses the function symbols of the monomorphic signature ($S_\mathcal{F}$), the set of type symbols is the ($S_\mathcal{T}$) translation by t of all possible ground type term instances allowed by the monomorphic signature. The function type declarations ($\mathcal{F}$) are all possible ground type term instances of the type variables of each function symbols monomorphic signature's type declaration.

I first show that the translation tv is a bijection, that inequalities and unifiers are preserved and that the $\prec'$ ordering is a simplification order. Then I show the lifting lemmas as bisimulations of the type-symbol and monomorphic inferences.

Lemma 3.5.3.1 (Bijection of tv)**.**
For all monomorphic terms t it holds that $tv^{-1}(tv(t)) = t$.

Proof. By induction over t.

- If t is a variable u^τ then $tv^{-1}(tv(u^\tau)) \Rightarrow tv^{-1}(u^{ty(\tau)}) \Rightarrow u^{ty^{-1}(ty(\tau))}$. Since $ty^{-1}(ty(\tau)) = \tau$ by construction, the property holds for variables.

- If t is a term of the structure $f\langle\tau_r,\,\tau_1,\,\ldots,\,\tau_n\rangle(t_1,\,\ldots,\,t_m)$ then
  $tv^{-1}(tv(f\langle\tau_r,\,\tau_1,\ldots,\tau_n\rangle(t_1,\ldots,t_m))) \Rightarrow$
  $tv^{-1}(f\langle ty(\tau(t_1)),\,\ldots,\,ty(\tau(t_m)) \to ty(\tau_r),\,ty(\tau_1),\,\ldots,\,ty(\tau_n)\rangle(tv(t_1),\,\ldots,\,tv(t_m))) \Rightarrow$
  $f\langle ty^{-1}(ty(\tau_r)),\,ty^{-1}(ty(\tau_1)),\,\ldots,\,ty^{-1}(ty(\tau_n))\rangle(tv^{-1}(tv(t_1)),\,\ldots,\,tv^{-1}(tv(t_m)))$
  which is by induction hypothesis ($tv^{-1}(tv(t_i)) = t_i$)
  $f\langle ty^{-1}(ty(\tau_r)),\,ty^{-1}(ty(\tau_1)),\,\ldots,\,ty^{-1}(ty(\tau_n))\rangle(t_1,\,\ldots,\,t_m)$
  by the fact that ty is bijective by construction
  $f\langle\tau_r,\,\tau_1,\,\ldots,\,\tau_n\rangle(t_1,\,\ldots,\,t_m)$

□

In order for $\prec'$ to be total on ground terms, we need that (ground) terms which are not equal before the translation tv^{-1} are not equal afterwards.

Lemma 3.5.3.2 (Terms Stay Distinct)**.**
For all type-symbol terms t and s it holds that if $t \neq s$ then $tv^{-1}(t) \neq tv^{-1}(s)$.

Proof. By induction over t and s. If both t and s are distinct variables before translation, then both are distinct variables after translation. If t (s) is a variable and s (t) a non-variable term, then $tv^{-1}(t)$ ($tv^{-1}(s)$) is still a variable and $tv^{-1}(s)$ ($tv^{-1}(t)$) still a non-variable term. If both t and s are non-variable terms then either

- Their top function symbol is different. It is not changed and therefore it is still different after translation.
- Their top function symbol is equal, but the return type is different. Then $\tau(f, \tau_1, \ldots, \tau_n)$ is different and thus they have a different type symbol after translation.
- A subterm is different, then by induction hypothesis those subterms are different after translation.

□

In the bisimulation lemmas below we require that terms that are unifiable before the translation tv are also unifiable after. Therefore, I show this fact now.

Lemma 3.5.3.3 (Unifiers are Preserved by Translation)**.**
The translation tv preserves unifier: $tv(t\{u_1 \mapsto s_1, \ldots\}) = tv(t)\{tv(u_1) \mapsto tv(s_1), \ldots\}$.

Proof. The translation $tv(u)$ of a variable u is still a variable, so $\{tv(u_1) \mapsto tv(s_1), \ldots\}$ is still a substitution. The remaining proof is by induction on the rules of tv.

- Clearly, the variable case holds: $tv(u\{u \mapsto s, \ldots\}) = tv(u)\{tv(u) \mapsto tv(s), \ldots\}$.
- For the function case, $tv(f\langle\tau_1, \ldots, \tau_n\rangle(t_1, \ldots, t_m)\{u_1 \mapsto s_1, \ldots\}) \doteq$ $f\langle\tau(f, \tau_1, \ldots, \tau_n)\rangle(tv(t_1), \ldots, tv(t_m))\{tv(u_1) \mapsto tv(s_1), \ldots\}$. Since f is a function symbol and $\tau_1, \ldots, \tau_n$ are ground, they are unaffected by substitutions. Thus, the equality depends only on the subterms t_i. From the induction hypothesis we know for each t_i that $tv(t_i\{u_1 \mapsto s_1, \ldots\}) = tv(t_i)\{tv(u_1) \mapsto tv(s_1), \ldots\}$.

□

For the type-symbol superposition calculus to work it needs as a parameter a simplification ordering, therefore, I have to show that the translated ordering is still one.

Lemma 3.5.3.4 ($\prec'$ is a Simplification Ordering)**.**
If $\prec$ is a simplification ordering for monomorphic first-order logic, the derived ordering $\prec'$ for type-symbol first-order logic is also a simplification ordering.

Proof. Suppose the lemma does not hold, then $\prec'$ is either not ground total, not irreflexive, not transitive, not compatible with Σ-operations or not closed under substitutions. In the following, let s'_1, s'_2 and s'_3 be three type-symbol ground terms and s_1 be $tv^{-1}(s'_1)$, s_2 be $tv^{-1}(s'_2)$ and s_3 be $tv^{-1}(s'_3)$ be their monomorphic translation.

1. If $\prec'$ is not ground total, then there must be a ground s'_1 and a ground s'_2 such that $s'_1 \neq s'_2$ and neither $s'_1 \prec' s'_2$ nor $s'_2 \prec' s'_1$. From Lemma 3.5.3.2 I know that $s_1 \neq s_2$ and I know that s_1 and s_2 are ground. By applying the definition of $\prec'$ I get that neither $s_1 \prec s_2$ nor $s_2 \prec s_1$ can then hold. But then $\prec$ cannot be a simplification ordering.

2. If $\prec'$ is not irreflexive then there must be an s'_1 such that $s'_1 \prec' s'_1$. By applying the definition of $\prec'$ I get that there must also be an s_1 such that $s_1 \prec s_1$. But then $\prec$ cannot be a simplification ordering.

3. If $\prec'$ is not transitive then there must be an s'_1, an s'_2 and s'_3 such that $s'_1 \prec' s'_2$ and $s'_2 \prec' s'_3$ but not $s'_1 \prec' s'_3$. By applying the definition of $\prec'$ I get that there must also be an s_1, an s_2 and s_3 such that $s_1 \prec s_2$ and $s_2 \prec s_3$ but not $s_1 \prec s_3$. But then $\prec$ cannot be a simplification ordering.

4. If $\prec'$ is not compatible with Σ-operations then there must be an s'_1 and an s'_2 such that $s'_1 \prec' s'_2$ does not imply $f(t_1,\ldots,s'_1,\ldots,t_m) \prec' f(t_1,\ldots,s'_2,\ldots,t_m)$. By applying the definition of $\prec'$ there must be an s_1 and s_2 such that $s_1 \prec s_2$ does not imply $f(t_1,\ldots,s_1,\ldots,t_m) \prec f(t_1,\ldots,s_2,\ldots,t_m)$. But then $\prec$ cannot be a simplification ordering.

5. If $\prec'$ is not closed under substitutions then there must be a $\sigma' = \{u_1 \mapsto t'_1,\ldots,u_n \mapsto t'_n\}$, s'_1 and s'_2 such that $s'_1 \prec' s'_2$ does not imply $s'_1\sigma' \prec' s'_2\sigma'$. Let σ be $\{u_1 \mapsto tv^{-1}(t'_1),\ldots,u_n \mapsto tv^{-1}(t'_n)\}$, since u_i and t'_i are of the same type symbol, this is well-typed as well. Then there must be an s_1 and s_2 such that $s_1 \prec s_2$ does not imply $s_1\sigma \prec s_2\sigma$. But then $\prec$ cannot be a simplification ordering.

6. If $\prec'$ does not have the subterm property, then there must be an s'_1 and an s'_2 such that s'_2 is a proper subterm of s'_1 but it does not hold that $s'_2 \prec' s'_1$. By applying the definition of $\prec'$ there must be an s_1 and s_2 such that it does not hold that $s_2 \prec s_1$. But then $\prec$ cannot be simplification ordering.

□

To show that both the monomorphic superposition calculus and the type-symbol superposition calculus are bisimulations of each other, I show that each of their inferences can simulate the corresponding inference of the other calculus. I start with the Equality Resolution and the Equality Factoring inferences and then show the Negative Superposition and the Positive Superposition inferences.

Lemma 3.5.3.5 (Bisimulation Lemma for the Equality Resolution and Equality Factoring Inferences)**.** Let C be a monomorphic clause and $m2ts(C)$ the translated clause in type-symbol first-order logic. Then there is an equality resolution (or equality factoring) inference of $m2ts(C)$ if and only if there is an equality resolution (or equality factoring) inference from C.

Proof. Suppose there is an $m2ts(C)$ so that there is an equality resolution inference for $m2ts(C)$, such that the inference result is not a result of the equality resolution inference from C (or vice versa). Then there must be at least one side condition of the equality resolution inference that is fulfilled for $m2ts(C)$, but not for C (or vice versa). The side conditions for the equality resolution inference (Sect. 3.3) are:

1. σ mgu of s and s'
2. a) $(s \not\approx s')\sigma$ is maximal in $(C' \vee s \approx s')\sigma$ and no literal is selected or
   b) $s \not\approx s'$ is the selected literal.

I now show why each side condition must also be fulfilled for C if it is fulfilled for $m2ts(C)$ and fulfilled for $m2ts(C)$ if it is fulfilled for C.

1. The translation preserves the most general unifier (Lemma 3.5.3.3).
2. a) $(s \not\approx s')$ is maximal in $(C' \vee s \approx s')$ if and only if it is maximal in the translated clause, because translated literals are compared by translating them back to type-symbol and comparing them there.
   b) $s \not\approx s'$ is the selected literal either in both or in neither because the same selection of literals is used for $m2ts(C)$ and C.

Thus, the equality resolution inference is possible for $m2ts(C)$ if and only if the type-symbol inference is possible with C.

Suppose there is a ground $m2ts(C)$ so that there is an equality factoring inference for $m2ts(C)$, such that the inference result is not a result of the equality factoring inference from C. Then there must be at least one side condition of the equality factoring inference that is fulfilled for $m2ts(C)$, but not for C. The side conditions for the equality factoring inference (Sect. 3.3) are:

1. $s\sigma = s'\sigma$, where σ is the most general unifier of s and s'.
2. $(s \approx t)\sigma$ maximal in $(C' \vee s' \approx t' \vee s \approx t)\sigma$ and no literal is selected.
3. $s'\sigma \not\preceq t'\sigma$ and $s\sigma \not\preceq t\sigma$

I now show why each side condition must also be fulfilled for C if it is fulfilled for $m2ts(C)$ (and vice versa).

1. The translation preserves the most general unifier (Lemma 3.5.3.3).
2. The orderings and selection are the same, because the ordering of C is used for $m2ts(C)$.
3. The orderings are the same, because the ordering of C is used for $m2ts(C)$.

Thus, the equality factoring inference is possible for $m2ts(C)$ if and only if the type-symbol inference is possible with C. □

Lemma 3.5.3.6 (Bisimulation Lemma for the Superposition Inferences)**.**
Let $C = C' \vee s \approx s'$ and $D = D' \vee t \approx t'$ be two monomorphic clauses. There is a superposition inference between $m2ts(C)$ and $m2ts(D)$ where $m2ts(s)$ and some subterm of $m2ts(t)$ if and only if there is a superposition inference between C and D.

Proof. Suppose there is an $m2ts(C)$ and $m2ts(D)$ so that there is a positive superposition inference for $m2ts(C)$ and $m2ts(D)$ such that the inference result is not a result of the positive superposition inference from C and D (or vice versa). Then there must be at least one side condition of the positive superposition inference that is fulfilled for $m2ts(C)$ and $m2ts(D)$ but not for C and D (or vice versa). The side conditions for the positive superposition inference (Sect. 3.3) are:

1. $s\sigma \not\preceq s'\sigma$ and $t\sigma \not\preceq t'\sigma$
2. s_2 is not a term variable
3. σ mgu of s and s_2
4. $(t \approx t')\sigma$ is strictly maximal in $(D' \vee t \approx t')\sigma$ and no literal is selected
5. $(s \approx s')\sigma$ is strictly maximal in $(C' \vee s \approx s')\sigma$ and no literal is selected

I now show why each side condition must also be fulfilled for C and D if it is fulfilled for $m2ts(C)$ and $m2ts(D)$ (and vice versa)

1. $s\sigma \not\preceq s'\sigma$ and $t\sigma \not\preceq t'\sigma$ hold for both, because the orderings of the translated and original terms is the same.
2. The translation preserves term variables, it never exchanges them to or from non-variable terms.
3. The translation preserves the most general unifier (Lemma 3.5.3.3).
4. The orderings and selection are the same, because the ordering of D is used for $m2ts(D)$.
5. The orderings and selection are the same, because the ordering of C is used for $m2ts(C)$.

Thus, the positive superposition inference is possible for C and D if and only if it is also possible for the type-symbol inference with $m2ts(C)$ and $m2ts(D)$.

Suppose there is a ground $m2ts(C)$ and $m2ts(D)$ so that there is a negative superposition inference for $m2ts(C)$ and $m2ts(D)$ such that the inference result is not a result of the negative superposition inference from C and D (or vice versa). Then there must be at least one side condition of the negative superposition inference that is fulfilled for $m2ts(C)$ and $m2ts(D)$ but not for C and D (or vice versa). The side conditions for the negative superposition inference (Sect. 3.3) are:

1. $s\sigma \not\preceq s'\sigma$ and $t\sigma \not\preceq t'\sigma$
2. s_2 is not a term variable
3. σ mgu of s and s_2
4. $(t \approx t')\sigma$ is strictly maximal in $(D' \vee t \approx t')\sigma$ and no literal is selected
5. a) $(s \not\approx s')\sigma$ is maximal in $(C' \vee s \approx s')\sigma$ and no literal is selected or

b) $s \not\approx s'$ is selected

I now show why each side condition must also be fulfilled for C and D if it is fulfilled for $m2ts(C)$ and $m2ts(D)$ (and vice versa). The side conditions and proofs 1. to 4. are identical to the side conditions and proofs of positive superposition inference.

5. a) $(s \not\approx s')\sigma$ is maximal in both, since the same ordering is used for translated and original literals.
   b) If $s \not\approx s'$ is selected in $m2ts(C)$ it is also selected in C, since the selection is the same for both clauses.

Thus, the negative superposition inference is possible for $m2ts(C)$ and $m2ts(D)$ if and only if it is also possible for the non-ground inference with C and D. □

Lemma 3.5.3.7 (Inference Results are Bisimular)**.**
Let R be the result of a monomorphic inference, then $m2ts(R)$ is the result of the corresponding translated inference.

Proof. The translation is bijective. From the bisimulation lemmas I now that there is always a corresponding inference for the translated and type-symbol clauses. Equality Resolution just removes a literal, so the remaining literals are by construction the translations of each other. Equality Factoring rearranges terms within two (in)equalities, since both the translated and the monomorphic inference rearrange the same terms, the result is still a translation of the other result. The Superposition inferences also rearrange terms, but within terms, still the same argument as for Equality Factoring applies. □

Before concluding refutational completeness for monomorphic first-order logic, I show that the encoding from monomorphic to type-symbol is satisfiability preserving. To this end I first show that a monomorphic interpretation can be translated to the type-symbol one and then the other direction. I do this by first creating a bijection between models for monomorphic and the translated type-symbol sets.

Let $(\mathcal{U}, \mathcal{D}, \mathcal{I}_\mathcal{T}, \mathcal{I}_\mathcal{F})$ be a monomorphic structure, then $tv_\mathcal{I}(\mathcal{U}, \mathcal{D}, \mathcal{I}_\mathcal{T}, \mathcal{I}_\mathcal{F})$ is a corresponding type-symbol structure for the problem translated with tv. It is defined as

$$tv_\mathcal{I}(\mathcal{U}, \mathcal{D}, \mathcal{I}_\mathcal{T}, \mathcal{I}_\mathcal{F}) = (\mathcal{U}, \mathcal{D}, \mathcal{I}^T_\mathcal{T}, \mathcal{I}^T_\mathcal{F})$$

where $\mathcal{I}^T_\mathcal{T}$ mappings are defined by

$$\mathcal{I}^T_\mathcal{T}(\kappa) = \mathcal{I}_\mathcal{T}(ty^{-1}(\kappa))$$

$\mathcal{I}^T_\mathcal{F}$ is function interpretations are defined by

$$f^\mathcal{I}_\tau(e_1, \ldots, e_m) = f^\mathcal{I}\langle \mathcal{I}_\emptyset(\tau_r), \mathcal{I}_\emptyset(\tau_1), \ldots, \mathcal{I}_\emptyset(\tau_n)\rangle(e_1, \ldots, e_m)$$

where $\tau = \kappa_1,\ldots,\kappa_m \to \kappa,\ \kappa_{a_1}, \ldots, \kappa_{a_n}$ and $\tau_r = ty^{-1}(\kappa)$, $\tau_1 = ty^{-1}(\kappa_{a_1}), \ldots, \tau_n = ty^{-1}(\kappa_{a_n})$

To show monomorphic and the translated type-symbol clause sets to be equisatisfiable, I first show that the terms (with the same/translated context) evaluate identical. Then I show that the clause sets are equisatisfiable. First, I start with the translation of terms from monomorphic to type symbol.

Lemma 3.5.3.8 (Monomorphic to Type Symbol Structure - Terms)**.**
Let $\mathcal{I}$ be a monomorphic interpretation with the monomorphic structure $(\mathcal{U}, \mathcal{D}, \mathcal{I}_{\mathcal{T}}, \mathcal{I}_{\mathcal{F}})$ of a monomorphic clause set N. For any monomorphic variable valuation $\mathcal{V}$, let a type-symbol variable valuation be defined as $\mathcal{V}^T(u_{ty(\tau)}) = \mathcal{V}(u_\tau)$. Then for the type-symbol interpretation $\mathcal{I}^T$ with the type-symbol structure $(\mathcal{U}^T, \mathcal{D}^T, \mathcal{I}^T_{\mathcal{T}}, \mathcal{I}^T_{\mathcal{F}}) = tv_{\mathcal{I}}(\mathcal{U}, \mathcal{D}, \mathcal{I}_{\mathcal{T}}, \mathcal{I}_{\mathcal{F}})$ and any monomorphic variable valuation $\mathcal{V}$ such that $\mathcal{V}^T(u_{ty(\tau)}) = \mathcal{V}(u_\tau)$, it holds that for all monomorphic terms t, $\mathcal{I}_{\mathcal{V}}(t) = \mathcal{I}^T_{\mathcal{V}^T}(tv(t))$.

Proof. By induction over t.

- In the case that t is a variable u_τ.
Then $tv(u_\tau) = u_{ty(\tau)}$ and therefore $\mathcal{I}^T_{\mathcal{V}^T}(tv(t)) = \mathcal{I}^T_{\mathcal{V}^T}(u_{ty(\tau)})$. By the semantics of type-symbol language, it follows that $\mathcal{I}^T_{\mathcal{V}^T}(u_{ty(\tau)}) = \mathcal{V}^T(u_{ty(\tau)})$. By the definition of $tv_{\mathcal{I}}$, we have $\mathcal{I}^T_{\mathcal{T}}(\kappa) = \mathcal{I}_{\mathcal{T}}(ty^{-1}(\kappa))$ and because ty is a bijection we have $\mathcal{I}^T_{\mathcal{T}}(ty(\tau)) = \mathcal{I}_{\mathcal{T}}(\tau)$. Therefore, $\mathcal{V}^T(u_{ty(\tau)}) = \mathcal{V}(u_\tau)$ confirms to the interpretations. By the monomorphic semantics $\mathcal{V}(u_\tau) = \mathcal{I}_{\mathcal{V}}(u_\tau)$.

- In the case that t is a function term $f\langle\tau_r, \tau_1, \ldots, \tau_n\rangle(t_1, \ldots, t_m)$.
Then $tv(f\langle\tau_r, \tau_1, \ldots, \tau_n\rangle(t_1, \ldots, t_m) = f\langle\tau\rangle(tv(t_1), \ldots, tv(t_m))$ and therefore $\mathcal{I}^T_{\mathcal{V}^T}(tv(t)) = \mathcal{I}^T_{\mathcal{V}^T}(f\langle\tau\rangle(tv(t_1), \ldots, tv(t_m)))$. By the type-symbol semantics it follows that $\mathcal{I}^T_{\mathcal{V}^T}(f\langle\tau\rangle(tv(t_1), \ldots, tv(t_m))) = f^{\mathcal{I}}_\tau(\mathcal{I}^T_{\mathcal{V}^T}(tv(t_1)), \ldots, \mathcal{I}^T_{\mathcal{V}^T}(tv(t_m)))$. By induction hypothesis, we know that for each i it holds that $\mathcal{I}^T_{\mathcal{V}^T}(tv(t_i)) = \mathcal{I}_{\mathcal{V}}(t_i)$ and thus it holds that $f^{\mathcal{I}}_\tau(\mathcal{I}^T_{\mathcal{V}^T}(tv(t_1)), \ldots, \mathcal{I}^T_{\mathcal{V}^T}(tv(t_m))) = f^{\mathcal{I}}_\tau(\mathcal{I}_{\mathcal{V}}(t_1), \ldots, \mathcal{I}_{\mathcal{V}}(t_m))$. By the definition of tv, we know that there is a bijection between τ and $\langle\tau_r, \tau_1, \ldots, \tau_n\rangle$. From the definition of $tv_{\mathcal{I}}$, we know that $f^{\mathcal{I}}_\tau(\mathcal{I}_{\mathcal{V}}(t_1), \ldots, \mathcal{I}_{\mathcal{V}}(t_m)) = f^{\mathcal{I}}\langle\mathcal{I}_\emptyset(\tau_r), \mathcal{I}_\emptyset(\tau_1), \ldots, \mathcal{I}_\emptyset(\tau_n)\rangle(\mathcal{I}_{\mathcal{V}}(t_1), \ldots, \mathcal{I}_{\mathcal{V}}(t_m))$, Therefore and from the monomorphic semantics, it follows that $f^{\mathcal{I}}\langle\mathcal{I}_\emptyset(\tau_r), \mathcal{I}_\emptyset(\tau_1), \ldots, \mathcal{I}_\emptyset(\tau_n)\rangle$ $(\mathcal{I}_{\mathcal{V}}(t_1), \ldots, \mathcal{I}_{\mathcal{V}}(t_m)) = \mathcal{I}_{\mathcal{V}}(f\langle\tau_r, \tau_1, \ldots, \tau_n\rangle(t_1, \ldots, t_m))$.

□

Now, I continue with the translation of terms from type symbol to monomorphic.

Lemma 3.5.3.9 (Type Symbol to Monomorphic Structure - Terms)**.**
Let $\mathcal{I}^T$ be a type-symbol interpretation with the type-symbol structure $(\mathcal{U}^T, \mathcal{D}^T, \mathcal{I}^T_{\mathcal{T}}, \mathcal{I}^T_{\mathcal{F}})$ of a type-symbol clause set N' generated by the translation $m2ts$ from a monomorphic clause set N. For any type-symbol variable valuation $\mathcal{V}^T$, let a monomorphic variable valuation be defined as $\mathcal{V}(u_{ty^{-1}(\kappa)}) = \mathcal{V}^T(u_\kappa)$. Then, for any type-symbol variable valuation and for the monomorphic interpretation $\mathcal{I}$ with the monomorphic structure $(\mathcal{U}, \mathcal{D}, \mathcal{I}_{\mathcal{T}}, \mathcal{I}_{\mathcal{F}}) = tv^{-1}_{\mathcal{I}}(\mathcal{U}^T, \mathcal{D}^T, \mathcal{I}^T_{\mathcal{T}}, \mathcal{I}^T_{\mathcal{F}})$, it holds that for all type-symbol terms t, $\mathcal{I}_{\mathcal{V}}(tv^{-1}(t)) = \mathcal{I}^T_{\mathcal{V}^T}(t)$.

Proof. By induction over t.

- In the case that t is a variable u_κ.
Then $tv^{-1}(u_\kappa) = u_{ty^{-1}(\kappa)}$ and therefore, $\mathcal{I}_{\mathcal{V}}(tv^{-1}(t)) = \mathcal{I}_{\mathcal{V}}(u_{ty^{-1}(\kappa)})$. By the monomorphic semantics it follows that $\mathcal{I}_{\mathcal{V}}(u_{ty^{-1}(\kappa)}) = \mathcal{V}(u_{ty^{-1}(\kappa)})$. By the definition of $tv_{\mathcal{I}}$ we have $\mathcal{I}^T_{\mathcal{T}}(\kappa) = \mathcal{I}_{\mathcal{T}}(ty^{-1}(\kappa))$. Therefore, $\mathcal{V}(u_{ty^{-1}(\tau)}) = \mathcal{V}^T(u_\kappa)$ confirms to the interpretations. By the type-symbol semantics $\mathcal{V}^T(u_\kappa) = \mathcal{I}^T_{\mathcal{V}^T}(u_\kappa)$.

- In the case that t is a function term $f\langle\tau\rangle(t_1, \ldots, t_m)$.
  Then by construction of tv it holds that $tv^{-1}(f\langle\tau\rangle(t_1, \ldots, t_m) = f\langle\tau_r, \tau_1, \ldots, \tau_n\rangle(tv^{-1}(t_1), \ldots, tv^{-1}(t_m))$ and therefore $\mathcal{I}_{\mathcal{V}}(tv^{-1}(t)) = \mathcal{I}_{\mathcal{V}}(f\langle\tau_r, \tau_1, \ldots, \tau_n\rangle(tv^{-1}(t_1), \ldots, tv^{-1}(t_m)))$. By the monomorphic semantics it follows that $\mathcal{I}_{\mathcal{V}}(f\langle\tau_r, \tau_1, \ldots, \tau_n\rangle(tv^{-1}(t_1), \ldots, tv^{-1}(t_m))) = f^{\mathcal{I}}\langle\mathcal{I}_{\emptyset}(\tau_r), \mathcal{I}_{\emptyset}(\tau_1), \ldots, \mathcal{I}_{\emptyset}(\tau_n)\rangle(\mathcal{I}_{\mathcal{V}}(tv^{-1}(t_1)), \ldots, \mathcal{I}_{\mathcal{V}}(tv^{-1}(t_m)))$. By induction hypothesis we know that for each i it holds that $\mathcal{I}_{\mathcal{V}}(tv^{-1}(t_i)) = \mathcal{I}^T_{\mathcal{V}^T}(t_i)$ and thus it holds that $f^{\mathcal{I}}\langle\mathcal{I}_{\emptyset}(\tau_r), \mathcal{I}_{\emptyset}(\tau_1), \ldots, \mathcal{I}_{\emptyset}(\tau_n)\rangle(\mathcal{I}_{\mathcal{V}}(tv^{-1}(t_1)), \ldots, \mathcal{I}_{\mathcal{V}}(tv^{-1}(t_m))) =$
  $f^{\mathcal{I}}\langle\mathcal{I}_{\emptyset}(\tau_r), \mathcal{I}_{\emptyset}(\tau_1), \ldots, \mathcal{I}_{\emptyset}(\tau_n)\rangle(\mathcal{I}^T_{\mathcal{V}^T}(t_1), \ldots, \mathcal{I}^T_{\mathcal{V}^T}(t_m))$. From the construction of $tv_{\mathcal{I}}$, we know that $f^{\mathcal{I}}\langle\mathcal{I}_{\emptyset}(\tau_r), \mathcal{I}_{\emptyset}(\tau_1), \ldots, \mathcal{I}_{\emptyset}(\tau_n)\rangle(\mathcal{I}_{\mathcal{V}}(t_1), \ldots, \mathcal{I}_{\mathcal{V}}(t_m)) = f^{\mathcal{I}}_{\tau}(\mathcal{I}_{\mathcal{V}}(t_1), \ldots, \mathcal{I}_{\mathcal{V}}(t_m))$, because from the construction of tv we know that there is a bijection between τ and $\langle\tau_r, \tau_1, \ldots, \tau_n\rangle$. Therefore and from the type-symbol semantics, it follows that $f^{\mathcal{I}}_{\tau}\langle\tau\rangle(\mathcal{I}^T_{\mathcal{V}^T}(t_1), \ldots, \mathcal{I}_{\mathcal{V}^T}(t_m)) = \mathcal{I}^T_{\mathcal{V}^T}(f\langle\tau\rangle(t_1, \ldots, t_m))$.

□

After showing the lemmas for the terms, I can now show the respective clause sets to be equisatisfiable. Again, I start with the direction from monomorphic to type symbol.

Lemma 3.5.3.10 (Monomorphic to Type Symbol Structure)**.**
Let $\mathcal{I}$ be a monomorphic interpretation with the monomorphic structure $(\mathcal{U}, \mathcal{D}, \mathcal{I}_{\mathcal{T}}, \mathcal{I}_{\mathcal{F}})$ of a monomorphic clause set N. For any monomorphic variable valuation $\mathcal{V}$, let a type-symbol variable valuation be defined as $\mathcal{V}^T(u_{ty(\tau)}) = \mathcal{V}(u_\tau)$. Then, for any monomorphic variable valuation and for the type-symbol interpretation $\mathcal{I}^T$ with the type-symbol structure $(\mathcal{U}^T, \mathcal{D}^T, \mathcal{I}^T_{\mathcal{T}}, \mathcal{I}^T_{\mathcal{F}}) = tv_{\mathcal{I}}(\mathcal{U}, \mathcal{D}, \mathcal{I}_{\mathcal{T}}, \mathcal{I}_{\mathcal{F}})$, it holds that for all monomorphic clauses $C \in N$, $\mathcal{I}_{\mathcal{V}}(C) = \mathcal{I}^T_{\mathcal{V}^T}(m2ts(C))$.

Proof. By induction over the interpretation rules for the monomorphic clause C.

- The case $C = t \approx s$ follows from Lemma 3.5.3.8.
- In the case $C = \neg\phi$. Then $\mathcal{I}^T_{\mathcal{V}^T}(m2ts(\neg\phi)) = \mathcal{I}^T_{\mathcal{V}^T}(\neg m2ts(\phi))$. By induction hypothesis (and because the definition of the interpretation of $\neg$ are identical), it follows that $\mathcal{I}^T_{\mathcal{V}^T}(\neg m2ts(\phi)) = \mathcal{I}_{\mathcal{V}}(\neg\phi)$.
- In the case $C = \phi_l \vee \phi_r$. Then $\mathcal{I}^T_{\mathcal{V}^T}(m2ts(\phi_l \vee \phi_r)) = \mathcal{I}^T_{\mathcal{V}^T}(m2ts(\phi_l) \vee m2ts(\phi_r))$. By induction hypothesis (and because the definition of the interpretation of $\vee$ are identical), it follows that $\mathcal{I}^T_{\mathcal{V}^T}(m2ts(\phi_l) \vee m2ts(\phi_r)) = \mathcal{I}_{\mathcal{V}}(\phi_l \vee \phi_r)$.
- In the case $C = \forall u : \tau.\ \phi$. Then $\mathcal{I}^T_{\mathcal{V}^T}(m2ts(\forall u : \tau.\ \phi)) = \mathcal{I}^T_{\mathcal{V}^T}(\forall u : ty(\tau).\ m2ts(\phi))$. And by the type-symbol interpretation rules $\mathcal{I}^T_{\mathcal{V}^T}(\forall u : ty(\tau).\ m2ts(\phi)) = \min\limits_{e \in \mathcal{I}^T(ty(\tau))}(\mathcal{I}^T_{\mathcal{V}^T[u \mapsto e]}(m2ts(\phi)))$.
  Because ty is bijective and by construction of $tv_{\mathcal{I}}$, it holds that $\mathcal{I}^T(ty(\tau)) = \mathcal{I}^T(\tau)$ and in particular $\mathcal{V}^T(u_{ty(\tau)}) = \mathcal{V}(u_\tau)$ for each valuation of u. Thus, it holds that $\min\limits_{e \in \mathcal{I}^T(ty(\tau))}(\mathcal{I}^T_{\mathcal{V}^T[u \mapsto e]}(m2ts(\phi))) = \min\limits_{e \in \mathcal{I}(\tau)}(\mathcal{I}^T_{\mathcal{V}[u \mapsto e]}(m2ts(\phi)))$. Furthermore, by the induction hypothesis we have $\mathcal{I}^T_{\mathcal{V}[u \mapsto e]}(m2ts(\phi)) = \mathcal{I}_{\mathcal{V}[u \mapsto e]}(\phi)$ and thus $\min\limits_{e \in \mathcal{I}(\tau)}(\mathcal{I}^T_{\mathcal{V}[u \mapsto e]}(m2ts(\phi))) = \min\limits_{e \in \mathcal{I}(\tau)}(\mathcal{I}_{\mathcal{V}[u \mapsto e]}(\phi))$. By the monomorphic interpretation rules we have that
  $\min\limits_{e \in \mathcal{I}(\tau)}(\mathcal{I}_{\mathcal{V}[u \mapsto e]}(\phi)) = \forall u : \tau.\ \phi$.

□

And continue with the inverse direction from type symbol to monomorphic.

Lemma 3.5.3.11 (Type Symbol to Monomorphic Structure)**.**
Let $\mathcal{I}^T$ be a type-symbol interpretation with the type-symbol structure $(\mathcal{U}^T, \mathcal{D}^T, \mathcal{I}^T_{\mathcal{T}}, \mathcal{I}^T_{\mathcal{F}})$ of a clause set generated by the translation *m2ts*. For any type-symbol variable valuation $\mathcal{V}^T$, let a monomorphic variable valuation be defined as $\mathcal{V}(u_{ty^{-1}(\kappa)}) = \mathcal{V}^T(u_\kappa)$. Then for any type-symbol variable valuation and for the monomorphic interpretation $\mathcal{I}$ with the type-symbol structure $(\mathcal{U}, \mathcal{D}, \mathcal{I}_{\mathcal{T}}, \mathcal{I}_{\mathcal{F}}) = tv_{\mathcal{I}}^{-1}(\mathcal{U}^T, \mathcal{D}^T, \mathcal{I}^T_{\mathcal{T}}, \mathcal{I}^T_{\mathcal{F}})$, it holds that for all type-symbol clauses C, $\mathcal{I}_{\mathcal{V}}(m2ts^{-1}(C)) = \mathcal{I}^T_{\mathcal{V}^T}(C)$.

Proof. By induction over the interpretation rules for the type-symbol clause C.

- The case $C = t \approx s$ follows from Lemma 3.5.3.9.
- In the case $C = \neg\phi$. Then $\mathcal{I}_{\mathcal{V}}(m2ts^{-1}(\neg\phi)) = \mathcal{I}_{\mathcal{V}}(\neg m2ts^{-1}(\phi))$. By induction hypothesis (and because the definition of the interpretation of $\neg$ are identical), it follows that $\mathcal{I}_{\mathcal{V}}(\neg m2ts^{-1}(\phi)) = \mathcal{I}^T_{\mathcal{V}^T}(\neg\phi)$.
- In the case $C = \phi_l \vee \phi_r$. Then $\mathcal{I}_{\mathcal{V}}(m2ts^{-1}(\phi_l \vee \phi_r)) = \mathcal{I}_{\mathcal{V}}(m2ts^{-1}(\phi_l) \vee m2ts^{-1}(\phi_r))$. By induction hypothesis (and because the definition of the interpretation of $\vee$ are identical), it follows that $\mathcal{I}_{\mathcal{V}}(m2ts^{-1}(\phi_l) \vee m2ts^{-1}(\phi_r)) = \mathcal{I}^T_{\mathcal{V}^T}(\phi_l \vee \phi_r)$.
- In the case $C = \forall u : \kappa.\, \phi$. Then $\mathcal{I}_{\mathcal{V}}(m2ts^{-1}(\forall u : \kappa.\, \phi)) = \mathcal{I}_{\mathcal{V}}(\forall u : ty^{-1}(\kappa).\, m2ts^{-1}(\phi))$. And by the monomorphic interpretation rules $\mathcal{I}_{\mathcal{V}}(\forall u : ty^{-1}(\kappa).\, m2ts^{-1}(\phi)) =$ $\min\limits_{e \in \mathcal{I}(ty^{-1}(\kappa))}(\mathcal{I}_{\mathcal{V}[u\mapsto e]}(m2ts^{-1}(\phi)))$. Because *ty* is bijective and by construction of $tv_{\mathcal{I}}$ it holds that $\mathcal{I}(ty^{-1}(\kappa)) = \mathcal{I}^T(\kappa)$ and in particular $\mathcal{V}^T(u_\kappa) = \mathcal{V}(u_{ty^{-1}(\kappa)})$ for each valuation of u. Thus, it holds that $\min\limits_{e \in \mathcal{I}(ty^{-1}(\kappa))}(\mathcal{I}_{\mathcal{V}[u\mapsto e]}(m2ts^{-1}(\phi))) = \min\limits_{e \in \mathcal{I}^T(\kappa)}(\mathcal{I}_{\mathcal{V}[u\mapsto e]}(m2ts^{-1}(\phi)))$.

  Furthermore, by the induction hypothesis we have $\mathcal{I}_{\mathcal{V}[u\mapsto e]}(m2ts^{-1}(\phi)) = \mathcal{I}^T_{\mathcal{V}^T[u\mapsto e]}(\phi)$ and thus $\min\limits_{e \in \mathcal{I}^T(\kappa)}(\mathcal{I}_{\mathcal{V}[u\mapsto e]}(m2ts^{-1}(\phi))) = \min\limits_{e \in \mathcal{I}^T(\kappa)}(\mathcal{I}^T_{\mathcal{V}^T[u\mapsto e]}(\phi))$. By the type-symbol interpretation rules we have that $\min\limits_{e \in \mathcal{I}^T(\kappa)}(\mathcal{I}_{\mathcal{V}[u\mapsto e]}(\phi)) = \forall u : \kappa.\, \phi$.

□

By showing both directions, I can conclude that the translation is satisfiability preserving.

Lemma 3.5.3.12 (Monomorphic to Type Symbol Translation Satisfiabilty Preserving)**.**
Let N be a clause set of monomorphic first-order logic it is satisfiable if and only if the translated clause set N' in type-symbol first-order logic is satisfiable.

Proof. The interpretation rules for clause sets are identical. Furthermore, with the translation $tv_{\mathcal{I}}$, a monomorphic interpretation can be translated into a type-symbol one and with $tv_{\mathcal{I}}^{-1}$, a type-symbol interpretation can be translated into a monomorphic one in such a way that for the original and the translated clauses satisfiability is preserved (Lemma 3.5.3.10 and 3.5.3.11). □

Because I showed the translation to be satisfiability preserving and I showed the type-symbol superposition calculus and the translated one to be bisimulations of each other, I have shown completeness for monomorphic superposition.

Theorem 3.5.3.13 (Static Refutational Completeness)**.**
Superposition is refutationally complete for monomorphic first-order logic.

Proof. I have shown that the translations preserves satisfiability (Lemma 3.5.3.12) and I have shown that the monomorphic superposition and the translated type-symbol superposition are bisimulations of each other (Lemma 3.5.3.5 and Lemma 3.5.3.6). Therefore, the completeness of the type-symbol version of superposition (Lemma 3.5.2.8) is lifted to the monomorphic version of superposition. $\square$

3.5.4. Lifting to Polymorphism

Here, I present the finial step of the soundness and completeness proof. I lift the monomorphic language to the initially described polymorphic language with type classes. As in the previous completeness proofs, I will omit predicates and only consider clauses instead of formulas.

Refutational Completeness

I will now lift monomorphic first-order logic, proved refutationally complete in the previous section, to the polymorphic first-order logic with type classes. First, I show that the typing is local, i.e. if an argument term is replaced by different term of the same type the overall type of the term does not change. Then, I lift the inferences and show that the monomorphic and polymorphic versions are equisatisfiable.

Lemma 3.5.4.1 (Type Judgement)**.**
For all terms $t[s]$ where s is t or a subterm of t it holds that $\tau(s) = \tau(s') \implies \tau(t[s]) = \tau(t[s'])$.

Proof. The proof is by induction of t. If t is s and replaced by s' the lemma is trivial (i.e. if t is a variable or constant). If t is of the form $f\langle\tau_1,\ldots,\tau_n\rangle(t_1,\ldots,t_m)$ and one of its subterms t_i is replaced by t_i' and that subterm is still of the same type, i.e. $\tau(t_i) = \tau(t_i')$, then by the typing rule for function (fun, Def. 3.1.2.2) it holds that $\tau(t[s]) = \tau(t[t_i'])$. $\square$

An important property of first-order logic is that variables are closed under rewriting, i.e. if an instance of a variable can be rewritten to some term t then, that term t is also an instance of the variable. This property is useful for superposition, because it allows to eliminate superposition inferences below variable positions.

Lemma 3.5.4.2 (Variables are Closed Under Rewriting with $R_{C\theta}$)**.**
Let $C \in N$ and θ be a substitution such that $C\theta \in G_\Sigma(C)$ and x be a variable occurring in C then: If $x\theta \rightarrow_{R_{C\theta}} t$ then there exists a θ' such that $x\theta' = t$ and $y\theta' = y\theta$ for every variable $y \neq x$ and furthermore $C\theta' \in G_\Sigma(C)$.

Proof. From the definition of literals it follows that only terms of the same type symbol can be equal. $R_{C\theta}$ is formed from equations where left hand and right hand side are of the same type symbol and replacing subterms by other terms of the same type symbol does not change the overall type of a term (Lemma 3.5.4.1). Since a variable can be instantiated to any term of the type it is assigned to, the lemma holds. □

I also need to lift the inferences from the monomorphic case to the polymorphic one. First, for the Equality Resolution and the Equality Factoring inferences and for the Positive Superposition and the Negative Superposition inferences.

Lemma 3.5.4.3 (Lifting Lemma for the Equality Resolution and Equality Factoring Inferences)**.** Let C be a clause and let θ be a substitution of only type variables such that $C\theta$ contains no type variables. Then every equality resolution (or equality factoring) inference of $C\theta$ is an instance of an equality resolution (or equality factoring) inference from C.

Proof. Suppose there is a $C\theta$ so that there is an equality resolution inference for $C\theta$ such that the inference result is not a result of the equality resolution inference from C. Then there must be at least one side condition of the equality resolution inference that is fulfilled for $C\theta$, but not for C. The side conditions for the equality resolution inference (Sect. 3.3) are:

1. $s\sigma = s'\sigma$, where σ is the most general unifier of s and s'.
2. a) $(s \not\approx s')\sigma$ *maximal* in $(C' \vee s \approx s')\sigma$ and no literal is selected or
   b) $s \not\approx s'$ is the selected literal.

I now show why each side condition must also be fulfilled for C if it is fulfilled for $C\theta$.

1. If the inference is applicable for $C\theta$, then $s\theta$ must be equal to $s'\theta$. By the existence and uniqueness of the most general unifier (Lemma 3.1.3.12), there must be a most general unifier θ_g of s and s' such that $\theta = \theta_g\theta'$. But then this side condition is also fulfilled for C.
2. a) If $(s \approx s')\theta$ is maximal in $(C' \vee s \approx s')\theta$ and nothing is selected, then there is no literal L in C' such that $L\theta \succ (s \approx s')\theta$. Therefore, the uninstantiated literal is also maximal.
   b) If $s \not\approx s'$ is selected in $C\theta$ it is also selected in C.

Thus, if the equality resolution inference is possible for $C\theta$, it is also possible for the inference with C.

Suppose there is a ground $C\theta$ so that there is a equality factoring inference for $C\theta$ such that the inference result is not a result of the equality factoring inference from C. Then there must be at least one side condition of the equality factoring inference that is fulfilled for $C\theta$, but not for C. The side conditions for the equality factoring inference (Sect. 3.3) are:

1. $s\sigma = s'\sigma$, where σ is the most general unifier of s and s'.
2. $(s \approx t)\sigma$ maximal in $(C' \vee s' \approx t' \vee s \approx t)\sigma$ and no literal is selected.
3. $s'\sigma \not\preceq t'\sigma$ and $s\sigma \not\preceq t\sigma$

I now show why each side condition must also be fulfilled for C if it is fulfilled for $C\theta$.

1. If the inference is applicable for $C\theta$, then $s\theta$ must be equal to $s'\theta$. By the existence and uniqueness of the most general unifier, there must be a most general unifier θ_g of s and s' such that $\theta = \theta_g\theta'$. But then this side condition is also be fulfilled for C.
2. Since $(s \approx t)\theta$ is maximal in $(C' \vee s' \approx t' \vee s \approx t)\theta$ and no literal is selected, there is no literal L in C' such that $L\theta \succ (s \approx t)\theta$. Therefore, the uninstantiated literal is also maximal. No literal is selected in C if none is selected in $C\theta$, thus this side condition is also fulfilled for C.
3. To be fulfilled $s'\sigma \not\preceq t'\sigma$, $s'\theta$ and $t'\theta$ can either be incomparable, equal or less. If they are incomparable, then with substitution stability s' and t' must also be incomparable. If they are equal, then s' and t' are either also equal or they are incomparable, otherwise substitution stability cannot hold. In the case where $s'\theta$ is less, then $t'\theta$, s' must either be less than t' or incomparable, again otherwise substitution stability cannot hold. The same argument holds for $s\sigma \not\preceq t\sigma$.

Thus, if the equality factoring inference is possible for $C\theta$ it is also possible for the non-ground inference with C. □

Lemma 3.5.4.4 (Lifting Lemma for the Superposition Inferences)**.**
Let $C = C' \vee s \approx s'$ and $D = D' \vee t \approx t'$ be two clauses without common variables and let θ be a substitution of only type variables such that $C\theta$ and $D\theta$ contain no type variables. If there is a superposition inference between $C\theta$ and $D\theta$ where $s\theta$ and some subterm of $t\theta$ are overlapped and $s\theta$ does not occur in $t\theta$ at or below a term variable position of t, then the inference is an instance of a superposition inference from C and D.

Proof. Suppose there is a $C\theta$ and $D\theta$ so that there is a positive superposition inference for $C\theta$ and $D\theta$ such that the inference result is not a result of the positive superposition inference from C and D. Then there must be at least one side condition of the positive superposition inference that is fulfilled for $C\theta$ and $D\theta$, but not for C and D. The side conditions for the positive superposition inference (Sect. 3.3) are:

1. $s\sigma \not\preceq s'\sigma$ and $t\sigma \not\preceq t'\sigma$
2. s_2 is not a term variable
3. $s\sigma = s_2\sigma$ and σ is the most general unifier of s and s_2
4. $(t \approx t')\sigma$ is strictly maximal in $(D' \vee t \approx t')\sigma$ and no literal is selected
5. $(s \approx s')\sigma$ is strictly maximal in $(C' \vee s \approx s')\sigma$ and no literal is selected

I now show why each side condition must also be fulfilled for C and D if it is fulfilled for $C\theta$ and $D\theta$.

1. To be fulfilled, $s\sigma \not\preceq s'\sigma$, $s\theta$ and $s'\theta$ can either be incomparable, equal or less. If they are incomparable, then with substitution stability s and s' must also be incomparable. If they are equal, then s and s' are either also equal or they are incomparable, otherwise substitution stability cannot hold. In the case where $s\theta$ is less, then $s'\theta$, s must either be less than s' or incomparable, again otherwise substitution stability cannot hold. The same argument holds for $t\sigma \not\preceq t'\sigma$.

2. That the inference is not at or below a variable position is part of the assumptions of the lemma.
3. If the ground inference is applicable for $C\theta$, then $s\theta$ must be equal to $s_2\theta$. By the existence and uniqueness of the most general unifier, there must be a most general unifier θ_g of s and s_2 such that $\theta = \theta_g\theta'$. But then this side condition is also be fulfilled for C and D.
4. If $(t \approx t')\theta$ is strictly maximal in $(D' \vee t \approx t')\theta$ and no literal is selected, then there is no literal L in D' such that $L\theta \succeq (t \approx t')\theta$. Therefore, there is also no L in D such that $L \succeq (t \approx t')$, otherwise the ordering would not be closed under substitutions and $(t \approx t')\theta$ could not be strictly maximal. Thus, $t \approx t'$ is strictly maximal and no literal is selected.
5. If $(s \approx s')\theta$ is maximal in $(C' \vee s' \approx s')\theta$ and nothing is selected, then there is no literal L in C' such that $L\theta \succ (s \approx t)\theta$, because the ordering is closed under substitutions. Therefore, there is also no L in D such that $L \succ (t \approx t')$, otherwise the ordering would not be closed under substitutions.

Thus, if the positive superposition inference is possible for $C\theta$ and $D\theta$ it is also possible for the non-ground inference with C and D.

Suppose there is a $C\theta$ and $D\theta$ so that there is a negative superposition inference for $C\theta$ and $D\theta$ such that the inference result is not a result of the negative superposition inference from C and D. Then there must be at least one side condition of the negative superposition inference that is fulfilled for $C\theta$ and $D\theta$ but not for C and D. The side conditions for the negative superposition inference (3.3) are:

1. $s\sigma \not\preceq s'\sigma$ and $t\sigma \not\preceq t'\sigma$
2. s_2 is not a term variable
3. $s\sigma = s_2\sigma$ and σ is the most general unifier of s and s_2
4. $(t \approx t')\sigma$ is strictly maximal in $(D' \vee t \approx t')\sigma$ and no literal is selected
5. a) $(s \not\approx s')\sigma$ is maximal in $(C' \vee s \approx s')\sigma$ and no literal is selected or
   b) $s \not\approx s'$ is selected

I now show why each side condition must also be fulfilled for C and D if it is fulfilled for $C\theta$ and $D\theta$. The side conditions and proofs 1. to 4. are identical to the side conditions and proofs of positive superposition inference.

5. a) If $(s \approx s')\theta$ maximal in $(C' \vee s' \approx s')\theta$ and nothing is selected, then there is no literal L in C' such that $L\theta \succ (s \approx t)\theta$. Therefore, there is also no L in D such that $L \succ (t \approx t')$, otherwise the ordering would not be closed under substitutions.
   b) If $s \not\approx s'$ is selected in $C\theta$ it is also selected in C.

Thus, if the negative superposition inference is possible for $C\theta$ and $D\theta$ it is also possible for the non-ground inference with C and D. □

Let N be a set of clauses, then $G^{\tau}_{\Sigma}(N)$ is defined as the union of all type-ground (monomorphic) instances of all clauses of N. I have also shown that a set of polymorphic clauses with type variables is satisfiable if and only if its monomorphic instances are satisfiable. First, by lifting refutations from the monomorphic to the polymorphic (plus type classes) setting. Then, by lifting models from the monomorphic to the polymorphic (plus type classes) setting.

Lemma 3.5.4.5 (Lifting)**.**
Let N be a set of clauses. Then $\bot \in \text{Sup}^*(G^{\tau}_{\Sigma}(N))$ implies $\bot \in \text{Sup}^*(N)$.

Proof. Because of the lifting lemmas (Lemma 3.5.4.3 and 3.5.4.4), each step of the monomorphic proof (in $\text{Sup}^*(G^{\tau}_{\Sigma}(N))$) is mirrored in the polymorphic clause set ($\text{Sup}^*(N)$). □

Lemma 3.5.4.6 (Model Lifting)**.**
Let N be a countable set of clauses. Then $\mathcal{I} \vDash G^{\tau}_{\Sigma}(N)$ implies $\mathcal{I} \vDash N$.

Proof. The lemma holds if it holds for the individual clauses $C \in N$. The proof is by induction over the number of type variables I show $\mathcal{I} \vDash G^{\tau}_{\Sigma}(C)$ implies $\mathcal{I} \vDash C$.

- If C does not contain any type variables, then $C \in G^{\tau}_{\Sigma}(C)$ and thus $\mathcal{I} \vDash C$.
- Otherwise, let α be a type variable occurring in C, i.e. $C = \forall_{\tau}\ \alpha : K.\ C'$ and let $K \neq \emptyset$. Because $\mathcal{I} \vDash G^{\tau}_{\Sigma}(C)$ and by induction hypothesis we know that $\mathcal{I} \vDash C'\{\alpha \mapsto \tau\}$ for all ground instances τ of α. From Corollary 3.1.4.14, we know that $\mathcal{I}(K) = \{\mathcal{I}(\tau) \mid \tau : K \text{ and } \tau \text{ is ground}\}$, i.e. the interpretation of a type-class constraint is exactly the union of the interpretations of the ground type terms that are in that type-class constraint. Thus $\min_{d \in \mathcal{I}(K)} (\mathcal{I}_{\emptyset[\alpha \to d],\emptyset}(C')) = 1$ and therefore $\mathcal{I} \vDash C$.
- Finally, let α be a type variable occurring in C, i.e. $C = \forall_{\tau}\ \alpha : K.\ C'$ and let $K = \emptyset$. Because $\mathcal{I} \vDash G^{\tau}_{\Sigma}(C)$ and by induction hypothesis we know that $\mathcal{I} \vDash C'\{\alpha \mapsto \tau\}$ for all ground instances τ of α, which are all possible ground terms (because $K = \emptyset$). If there is a domain $d \in \mathcal{D}$ such that the is no ground type term τ such that $\mathcal{I}_{\emptyset}(\tau) = d$, then the interpretation is still a model with the set of domains replaced by $\mathcal{D}' = \mathcal{D} \backslash \{d\}$. With $\mathcal{D}'$, the interpretation of the ground instances of α is an interpretation of all its domains and thus $\mathcal{I} \vDash C$.

□

Because both the refutations and the models can be lifted from the monomorphic to the polymorphic (plus type classes) setting, I can conclude that both the monomorphic and the polymorphic setting are equisatisfiable.

Lemma 3.5.4.7 (Polymorph and Monomorph are Equisatisfiable)**.**
A countable set N of Σ-clauses is satisfiable if and only if its monomorphic version $G^{\tau}_{\Sigma}(N)$ is satisfiable.

Proof. By case distinction if $G^{\tau}_{\Sigma}(N)$ is satisfiable.

- Suppose $G^{\tau}_{\Sigma}(N)$ does not have a model, then $\bot \in \text{Sup}^*(G^{\tau}_{\Sigma}(N))$. By lifting (Lemma 3.5.4.5) we have $\bot \in \text{Sup}^*(N)$ and thus N does not have a model.
- Suppose $G^{\tau}_{\Sigma}(N)$ has a model, then there is an interpretation $\mathcal{I}$ such that $\mathcal{I} \vDash G^{\tau}_{\Sigma}(N)$. By model lifting (Lemma 3.5.4.6) we also have $\mathcal{I} \vDash N$.

□

I have now shown that the polymorphic first-order language with type classes can be encoded into the monomorphic one by exhaustively instantiating the type variables. I have also shown that this preserves satisfiability. Therefore, the static completeness already shown for the monomorphic case can be lifted to the polymorphic case with type classes.

Theorem 3.5.4.8 (Static Refutational Completeness)**.**
Superposition is refutationally complete for polymorphic first-order logic with type variables.

Proof. I have shown that a countable set N of Σ-clauses is satisfiable if and only if its monomorphic version $G^{\tau}_{\Sigma}(N)$ is satisfiable (Lemma 3.5.4.7). Furthermore, I have shown that a proof in the monomorphic version is lifted to the polymorphic version (Lemma 3.5.4.5). Therefore, the completeness of the monomorphic version of superposition (Lemma 3.5.3.13) is lifted to the polymorphic (with type classes) version of superposition. $\square$

In the previous lifting steps I stopped with static refutational completeness, because they were intermediate steps. The refutational completeness of superposition for the polymorphic first-order language with type classes is the final step. Therefore, I also show that dynamic refutational completeness holds. The difference is that static refutational completeness states that any saturated set of clauses that does not contain the empty clause is satisfiable, whereas dynamic completeness states that for any (fair) sequence of inferences that the initial clause set is satisfiable if and only if all intermediate clause sets do not contain the empty clause.

Lemma 2.2.4.12, which states that a fair run has a saturated limit, also holds for the superposition calculus for polymorphic first-order logic with type classes.

Lemma 3.5.4.9 (Limit is saturated)**.**
If a run is fair, then its limit is saturated up to redundancy.

Proof. If the limit is not saturated up to redundancy, there must be an inference that can still be applied. Thus, both its premises must be in N_*, but the result is not in N_* and is not redundant with respect to N_*. Such an inference must be enabled in N_* and thus must be enabled in all j that are $j \geq i$ for some i. By definition, such a run cannot be fair. $\square$

An important property of first-order logic is compactness, i.e. a clause set is unsatisfiable if and only if a finite subset of that clause set is unsatisfiable. In particular, the property implies that all proofs are finite. This property also holds for my polymorphic first-order logic with type classes.

Lemma 3.5.4.10 (Compactness)**.**
Let N be a set of polymorphic first-order logic first-order formulas with type classes. Then N is unsatisfiable if and only if some finite subset of N is unsatisfiable.

Proof. $\Leftarrow$ If a finite subset is unsatisfiable the whole set is unsatisfiable. $\Rightarrow$ Let N be unsatisfiable. Then $\mathit{Sup}^*(N)$ is also unsatisfiable and by refutational completeness I have $\bot \in \mathit{Sup}^*(N)$. Because of the well-foundness constraint of the orderings the proof of $\bot$ (its inference derivation tree) is of finite size. Then the finite subset of N is the set of assumptions of the finite proof of $\bot$. $\square$

Finally, I can show that the superposition calculus for the polymorphic first-order language with type classes is refutationally complete also in the dynamic view. This is important because the static view does not lend itself to an implementation, while the dynamic view guarantees that any fair run (i.e. fair execution of an implementation) on an unsatisfiable clause set results in the discovery of the empty clause.

Theorem 3.5.4.11 (Dynamic Refutational Completeness)**.**
Let $N_0 \vdash N_1 \vdash N_2 \ldots$ be a fair run and N_* its limit. Then N_0 has a model if and only if $\bot \notin N_*$.

Proof. $\Rightarrow$ Obvious.
$\Leftarrow$ The run is fair and therefore N_* is saturated up to redundancy (Lemma 3.5.4.9). If $\bot \notin N_*$ and since we know that N_* is saturated up to redundancy it has to have a model (Theorem 3.5.4.8). Every clause in N_0 is an element of N_* or is redundant with respect to N_* and thus the model of N_* is also a model of N_0. $\square$

I have now shown static and dynamic completeness of superposition for the presented polymorphic language with type classes.

3.6. Related Work

The resolution calculus, invented by Robinson [52], first introduced saturation-based theorem proving. Resolution is refutationally complete for first-order logic, i.e. from any unsatisfiable set of clauses it can (eventually) derive the empty clause. Resolution has no build-in equality reasoning, but equality can be axiomatized. Reasoning with an axiomatization of equality can result in significant overhead. To overcome the overhead introduced by axiomatizating equality, Robinson and Wos introduced paramodulation [51]. An import goal for them was to increase "convergence" by reducing "unnecessary 'noise' in the proof-search space" by deriving no "side-effect" clauses which for example stem from the axiomatization of equality and reasoning with that axiomatization. The initial unrestricted paramodulation calculus fell short of this goal. Knuth and Bendix [36] showed that confluence of terminating rewrite systems (i.e. equational logic) can be tested by normalization via critical pairs. Lankford [40] restricted rewriting and critical pairs with the use of reduction orderings such that only larger sides of equations had to be considered. Hsiang and Rusinowitch introduced ordered paramodulation [32] a restriction of paramodulation where only maximal literals of clauses need to be considered for inferences. Bachmair and Ganzinger [4] introduced (strict) superposition, a restriction of ordered paramodulation in combination with insights from ordered rewriting that does not require inferences into smaller sides of equations.[3] They showed refutational completeness for first-order logic with equality by adding the equality factoring inference to the superposition calculus. Furthermore, they introduced the notion of (saturation up to) redundancy. Bachmair and Ganzinger [6] give a comprehensive overview of superposition, its variants and origins.

I differentiate between types and sorts, even though they have similar origins and the same goal of separating different domains. If domain membership is (easily) decidable, I will consider the separation of domains to be types. If domain membership is more difficult, e.g. defined via first-order predicates, I will consider the separation of domains to be sorts. Extreme examples of sorts allow arbitrary first-order formulas as sorts, e.g. those presented by Hailperin [27].

[3]Note that this is an approximation of only requiring them in larger sides.

Sorted first-order calculi and their completeness have been investigated dating back even to Herbrand [30]. Incidentally, according to Schmidt [55], Herbrand's completeness proof for his many-sorted calculus was incorrect. Schmidt then gave a monomorphic (many-sorted) calculus. Schmidt-Schauss [56] showed refutational completeness of an order-sorted calculus with resolution, factoring and unrestricted paramodulation. The order-sorted first-order logic is a many-sorted first-order logic extended with a subsort relation, such that each term is of a particular sort. Weidenbach [67] presented a general sort-system for superposition.

Programming languages generally favor types over sorts. An example is ML, for which Milner [45] introduced the concept of (parametric) polymorphism. Parametric polymorphism captures that one function can handle arguments of different types in the same way. Wadler and Blott [63] introduced type classes in Haskell, to allow functions to be defined in different ways for different types (i.e. overloading also called ad-hoc polymorphism). The type classes guarantee that certain functions are defined on each type that is contained in the type class. In general, type classes can declare not only functions but also additional properties these functions must have. Waldmann [64] shows that concepts of overloading and parametricity were not always treated sufficiently different in the literature of order-sorted systems. ML's polymorphism is now called ML-style polymorphism or, more formally, rank-1 polymorphism without phantom type variables but with let-polymorphism and an equality type class. Phantom type variables are type-variable arguments of predicates and functions that do not occur in any argument's type or in the return type. Rank-0 Polymorphism is monomorphism, rank-1 polymorphism allows outermost type quantification and rank-$n{+}1$ allows up to n nested occurrences of type quantifications in left-hand sides of the function type constructor.

Isabelle/HOL [49] is a proof assistant based on higher-order logic and functional programming. Isabelle is written mainly in ML and its type system is based on rank-1 polymorphism without phantom type variables. Nipkow [47] extended the polymorphism used in Isabelle/HOL by introducing order-sorted polymorphism. The extension allows types to be restricted by sorts, which allow subsorts. Haftmann and Wenzel [26] introduced constructive type classes, which combine the logic aspect of order-sorted polymorphism with the operative aspect of Haskell's type classes. In particular, this means that polymorphism with constructive type classes falls into my definition of type system (and not sort system). It is also the variant supported by my formalization. Isabelle is mostly based on declarative style, where functions and type classes are declared and not axiomatized. This enables Isabelle to enforce the type system to ensure that functions are used as declared and that properties of type classes are fulfilled. My formalization is based on first-order logic extended with a type system. This means its input is a set of first-order formulas (axioms). Integration of the type system into the calculus allows the use of the type system and thus removes the burden to axiomatize the restriction the type system provides. Isabelle's declarative functions nonetheless turn into first-order formulas, i.e. (conditional) equations.

The formalization of this chapter supports rank-1 polymorphism (including phantom type variables) extended with type classes. Interestingly enough, phantom type arguments for predicates are useful in the translation (of type classes) from Isabelle/HOL, even though they are not used in Isabelle/HOL. Pirate therefore implements phantom type arguments for predicates.

The origins of monomorphic type systems in automated theorem prover implementations can be found in Otter [44]. While Otter has no type system as such, it has support for multiple equality predicates. It is commonly known that using a different equality predicate for each type allows sound type erasure (in the absence of overloaded symbols). SNARK [61], which like Otter handles first-order logic with equality (via resolution and paramodulation), also supports unary-predicates as sorts and subsorts. This is similar to superposition-based SPASS, which also supports special handling of sorts. Most automated provers that support theories also support some form of monomorphic types. This is in particular the case for SMT solvers, which must separate the Int and Real domains from the remainder of the problem. Both Vampire [39] and our previous work on SPASS [9] support first-order logic with native monomorphic types. The native types are more efficient than an encoding into many-sorted sorts, which are an instance of the sorts available in SPASS.

There are also a few automated theorem provers that support stronger type systems. The SMT solver Alt-Ergo [13] implements ML-style polymorphism. An evaluation shows that its implementation of polymorphism does not scale well, i.e. while it is better than other encodings at 50 facts. with 500 facts its native implementation falls behind efficient encodings [11]. For 500 facts the success rate for polymorphism also falls behind the success rate for monomorphism. The superposition-based prover Zipperposition [58] implements rank-1 polymorphism, but no type classes. LEO-II [8] implements high-order logic with a prefix-polymorphism. It restricts polymorphism, because "full polymorphism adds many non-trivial choice points to the already challenging search space of LEO-II", which is caused by its higher-order setting. LEO-III's LeoPARD data structures [68] implement polymorphism without type constructors (except for function types). SMT solvers like Alt-Ergo are necessarily incomplete for first-order logic. LEO-II and LEO-III are also incomplete for first-order logic. As far as I am aware there is no refutational completeness proof for Zipperposition or its calculus.

In contrast to automated theorem provers, most proof assistants have a rich type system but generally require user interaction. The HOL family, e.g. HOL Light [28] and Isabelle/HOL, supports ML-style polymorphism. Proof assistants not only have rich type systems, but generally also a richer logic. The HOL family is based on higher order logic, instead of the first-order logic of many automated theorem provers. Proof assistants like Agda, Coq, Lean and Matita implement dependent types. Coq extends those with type classes which are implemented on top of the dependent types.

Translation tools from and to different logics or from verification problems to logics also have to deal with differences in type systems. Some of them introduce their own intermediate languages. Sledgehammer [10] uses a variety of encodings to get from Isabelle/HOL's higher-order polymorphic type system with type classes to one supported by an automated prover. For first-order provers it has to remove higher-order features like higher-order quantifications and lambdas. Type encodings generally use predicates (type guards) or additional term arguments (type tags) to encode type systems into less expressive ones. Using monotonicty analysis, many of the type guards or type tags can be omitted [11]. Boogie 2 [43] features its own intermediate language, which supports higher-ranked polymorphism. It translates its rich type language to untyped logic, with type guards and type tags, so that SMT solvers can handle it. Dafny [41] features its own imperative intermediate language, which supports polymorphism via generic classes. Dafny encodes its language into Boogie's to profit from automation provided by SMT solvers. Why3 [14, 15] has a ML-style polymorphic first-order language extended with algebraic data types and inductive predicates. It can encode the polymorphic language to monomorphic and untyped first-order logic in order to provided automation by using a variety of SMT and superposition-based tools. Its language is very close to the one we introduce here and in chapter 4. HOL(y)Hammer [34] encodes from HOL Light's polymorphic higher-order logic to first-order in order to profit from automation by SMT and superposition-based tools. It mainly uses encodings that are similar to those available in Sledgehammer. There is also recent work to encode dependently-typed systems into less expressive type systems, e.g. to encode F* to (monomorphic) SMT solvers [31]. Furthermore, there is ongoing work on building Sledgehammer-style tools for Coq [19].

The main standardized exchange formats for automated theorem provers are the TPTP syntax family and the SMT-LIB syntax. The TPTP syntax family defines a syntax which includes support for rank-1 polymorphic first-order and higher-order logic [12]. It does not support type classes and the ordering information of equations supported by our input syntax [9]. The SMT-LIB 2.5 standard [7] defines a monomorphic first-order logic which can have polymorphic theory declarations.

4. Induction

4.1. Introduction

Proving conjectures of functions over recursive data types is challenging, because it almost always requires proofs by induction. Automating such inductive proofs by structural induction raises four main issues. A *proof procedure* for non-inductive conjectures, in particular for the induction cases, is needed. This proof procedure must be extendable with induction. For a given conjecture a suitable *instance of the induction schema* must be selected. This suffices for easily automated conjectures. However, inductive proofs often require additional lemmas, which themselves require an inductive proof. Therefore, to automatically complete an inductive proof auxiliary propositions must be *guessed*. Since the guessed propositions can be non-theorems, a reliable way to *purge unprovable propositions* is useful.

My first contribution is the design of an automated inductive theorem prover that proposes solutions to all four challenges. Starting from the refutationally complete superposition calculus for polymorphic first-order logic with equality and type classes (Ch. 3), I introduce the SupInd calculus, an extension of superposition with structural induction over data types (Sect. 4.2). Just like superposition, SupInd requires first-order formulas to be transformed to first-order clauses and conjectures are proven by refuting their negation, i.e. by showing the negated conjecture together with the axioms is unsatisfiable. To this end, the conjecture is negated, Skolemized and clausified and the axioms are Skolemized and clausified. The resulting clause sets, one for the negated conjecture and one for the axioms, together with the (initially empty) set of induction hypotheses clauses form a triple, which I call a *simple state*. I keep the clause sets separate because each induction step requires the negated conjecture and the induction hypotheses to be separate. Each induction step requires proving several induction cases, where every induction case has to be shown. Successful automating inductive proofs requires to introduce additional helpful propositions and in SupInd I distinguish one particular case of such propositions: those propositions that imply a previous simple state. Such a proposition is an alternative way to prove that simple state and I call it a *strengthening* of that simple state. To be able to express nested induction and nested strengthenings I need a tree-like structure to express cases where the refutation of one subtree is sufficient and cases where all subtrees are to be refuted. I capture this by the notation of a *state*, which is either a simple state, an *or*-state or an *and*-state. A simple state is proven if its clause sets contain the empty clause, an *or*-state is proven if one of its substates is proven and an *and*-state is proven if all its substates are proven. SupInd's inferences operate on states. The approach preserves completeness for pure first-order logic (i.e. without data types) while supporting guessing and proving propositions, in particular strengthenings, and nested induction proofs of arbitrary depth.

In this chapter, function symbols start with lowercase letters (e.g. the Skolem function xs and the function minus) and universally quantified variables with uppercase letters (e.g. X and Ys).

One challenge in automating induction is to find a good instance of the induction schemata, i.e. finding a good (sub)term to which to apply induction. In the refutational setting of SupInd, finding a good instance of the structural induction schemata is reduced to finding a good Skolem function (of the conjecture) to apply induction to. SupInd's structural induction rule then generates a new state with the necessary induction cases from the combination of simple state and Skolem function. Instead of guessing which Skolem function is a good choice or exhaustively applying induction to all Skolem functions, I propose a heuristic that lets a prover determine which Skolem function should be chosen next (Sect. 4.3). Consider the negated conjecture for the associativity of addition over the natural numbers ($\mathbb{N}$), constructed by $\mathit{0}$ and the successor function s: $\mathsf{x}+(\mathsf{y}+\mathsf{z}) \not\approx (\mathsf{x}+\mathsf{y})+\mathsf{z}$, where addition is defined by $\mathit{0}+Y \approx Y$ and $\mathit{s}(X)+Y \approx \mathit{s}(X+Y)$. Since $+$ is defined in terms of its first argument, selecting induction over y or z does not immediately yield a refutation, but selecting x does. Often, the base cases are already an indication of which Skolem should be preferred. This insight yields a technique where all potential induction positions can be tested for being preferable by solving a single first-order clause set and analyzing the resulting (unsatisfiablity) proof. Furthermore, even if there is no such proof, i.e. if this clause set is (first-order) satisfiable, then, the base cases of induction of any of the Skolem functions are in general also (first-order) satisfiable without additional inductive lemmas. Therefore in this case, a successful proof needs search for additional propositions instead of further nested inductions.

Induction proofs often require additional lemmas which themselves can be proven by induction. Strengthening the conjecture is one way to discover such lemmas (Sect. 4.4). A standard technique to strengthen a conjecture is *generalization*, which replaces complex subterms of the conjecture by fresh variables. In SupInd, because of the refutational setting, generalization replaces complex subterms in the negated conjecture by fresh Skolem constants (Sect. 4.4.1). For example, let len be the length and rev be the reverse of a list, $-$ and $\leq$ be the familiar operations over $\mathbb{N}$, all specified axiomatically (in first-order logic). Let the conjecture be $\mathit{len}(Xs) \approx \mathit{len}(Ys) \rightarrow \mathit{len}(\mathit{rev}(Xs)) - \mathit{len}(Ys) \leq \mathit{len}(Ys)$ and the negated Skolemized conjecture, on which the SupInd calculus operates, be

$$\mathit{len}(\mathsf{xs}) \approx \mathit{len}(\mathsf{ys}) \land \mathit{len}(\mathit{rev}(\mathsf{xs})) - \mathit{len}(\mathsf{ys}) \not\leq \mathit{len}(\mathsf{ys})$$

Generalization strengthens this by replacing $\mathit{len}(\mathsf{ys})$ by a fresh Skolem y to

$$\mathit{len}(\mathsf{xs}) \approx \quad \mathsf{y} \quad \land \mathit{len}(\mathit{rev}(\mathsf{xs})) - \quad \mathsf{y} \quad \not\leq \quad \mathsf{y}$$

Induction on y results in a stronger induction hypothesis than induction over ys would have. Therefore, if the strengthened proposition holds, it tends to be easier to prove. Performing generalization on, for example, $\mathit{len}(xs)$ would yield a proposition that does not hold, because the dependency with $\mathit{len}(\mathit{rev}(xs))$ is lost. This is where my next contribution, extending generalization to exploit term dependencies, comes into play (Sect. 4.4.2). Generalization introduces a strengthened proposition by replacing subterms by a fresh Skolem constant for each syntactically different subterm that is replaced. The underlying idea of term-dependency preserving generalization is that multiple (syntactically different) subterms can be replaced by the same Skolem constant if those subterms can be shown to be equal. Thus where standard generalization computes a single proposition that implies the conjecture, term-dependency preserving generalization computes multiple propositions that together imply the conjecture. Term-dependency preserving generalization refines this idea by picking at least two super-terms of the same Skolem function (occurring in the negated conjecture) and then computes a generalization that is stronger than the standard generalization, by assuming that those super-terms are equal. The new propositions are then the stronger generalization and the proposition that those super-terms are in fact equal. Alone the stronger generalization might not imply the previous simple state, but together with the additional (equality) proposition it implies the previous simple state. For the example conjecture, term-dependency preserving generalization guesses the following two propositions:

$$(1)\ \mathit{rev}(Xs) \approx Xs \qquad (2)\ \mathit{len}(\mathit{rev}(Xs)) \approx \mathit{len}(Xs)$$

Both imply that $\mathit{len}(\mathsf{xs})$ equals $\mathit{len}(\mathit{rev}(\mathsf{xs}))$ and thus justify the term-dependency preserving generalization of the negated conjecture to

$$\mathsf{x} \approx \mathsf{y} \wedge \mathsf{x} - \mathsf{y} \not\leq \mathsf{y}$$

For SupInd to prove the initial conjecture directly by term-dependency preserving generalization, the generalized conjecture and either proposition (1) or proposition (2) must be proven, because one of the propositions must hold to justify the generalization step. Property (2) can be automatically proven by induction and first-order reasoning, without guessing further propositions. While SupInd can disprove property (1) by finding a counter example (the the two-element-list case) after three subsequent induction steps by reducing the negated conjecture to the axioms (Sect. 4.5).

The solver always simplifies as much as it can. After simplification the negated conjecture becomes

$$\mathsf{x} - \mathsf{x} \not\leq \mathsf{x}$$

Unfortunately even this conjecture cannot be proven, yet. This is because after induction over x, in the case where $\mathsf{x} \approx s(\mathsf{x}')$, the defining equations for subtraction $(-)$, in particular $s(X) - s(Y) \approx X - Y$, act as a "constructor sink". Because subtraction only occurs on the left-hand side, the successor constructors (s) disappear on the left-hand side of the $\leq$ predicate but not on the right-hand side. Hence, with each nested induction step, the negated conjecture accumulates a successor constructor (s) on the right-hand side of the $\leq$ predicate:

$\mathsf{x}' - \mathsf{x}'$	$\not\leq$	$s(\mathsf{x}')$	After one induction
$0 - 0$	$\not\leq$	$s(0)$	Base case ($\mathsf{x}' = 0$), trivial
$s(\mathsf{x}'') - s(\mathsf{x}'')$	$\not\leq$	$s(s(\mathsf{x}''))$	Step case ($\mathsf{x}' = s(\mathsf{x}'')$)
$\mathsf{x}'' - \mathsf{x}''$	$\not\leq$	$s(s(\mathsf{x}''))$	Step case (simplified)
	$\vdots$		

The induction hypothesis never matches the conjecture and a proof cannot be found without additional lemmas. My final contribution copes with constructor sinks by deriving a strengthened conjecture and side-conditions that do not contain the constructor sink term but imply the original conjecture (Sect. 4.4.3). The technique can be applied to any data type and conjecture, as long as a bound can be derived and all values up to that bound fulfill the conjecture. The underlying principle is the following: Let t be a term of the natural number data type and let u be an upper bound for t. A term u is an upper bound for t if all values of t are structurally smaller than those for u (Sect. 4.4.3). Then if $P(u)$ and $P(s(X)) \implies P(X)$ hold so must $P(t)$.

I call terms representing such upper bounds cap terms. Argument positions of function symbols that are cap term positions are discovered with the help of a syntactic criterion from the defining equations of every function symbol. In the example, the result of subtraction ($-$) must be (structurally) smaller or equal to its first argument and thus the first argument is a cap subterm. Applying this knowledge in the example yields two cases, the constructor-sink term replaced by its cap subterm ($\mathsf{x}-\mathsf{x}$ replaced by x in $\mathsf{x}-\mathsf{x} \not\leq \mathsf{x}$):

$$\mathsf{x} \not\leq \mathsf{x}$$

and the proposition that if the constructor-sink term holds with some bound, it holds for all n that are smaller, i.e. the proposition $s(N) \leq X \implies N \leq X$ and its negation:

$$s(\mathsf{n}) \leq \mathsf{x} \land \mathsf{n} \not\leq \mathsf{x}$$

With the presented techniques the initial conjecture can be automatically proven. Both induction cases can be refuted by nested induction and first-order reasoning alone. The strengthenings of the conjectures and the required property ($\mathit{len}(\mathit{rev}(Xs)) \approx \mathit{len}(Xs)$) for the term-dependency preserving generalization are (easily) provable using SupInd.

I have implemented my contributions in a tool called *Pirate* and evaluated it on the CLAM [33] and the IsaPlanner [22] benchmark suites (Sect. 4.7).

4.2. Superposition for Induction

Well-founded induction states that a property holds for all elements e, assuming that the property holds for all elements that are smaller (with respect to a well-founded relation) than e. Well-founded induction is called *(strong) structural induction*, when the well-founded relation is defined using the proper subterm relation. SupInd uses (strong) structural induction on algebraic data types such as lists (when represented as empty list (nil) and cons function (cons)) and the natural numbers (when represented as zero ($\mathit{0}$) and successor function (s)).

4.2.1. Typed First-Order Logic and Algebraic Data Types

The basis of the SupInd calculus is a polymorphic first-order language with type classes and data-type declarations. A type is an algebraic data type if it has a set of *constructor* function symbols that generate the elements of that type.

Syntax

A signature of the polymorphic first-order language with type classes and data types is a signature for the polymorphic first-order language with type classes (Sect. 3.1.1) extended by a set of data-type declarations $\mathcal{DT}$, declaring the data types.

Definition 4.2.1.1 (Signature). A signature for a polymorphic first-order language with type classes and data types is a tuple $\Sigma = (S_{\mathcal{F}},\, S_{\mathcal{P}},\, S_{\mathcal{T}},\, S_{\mathcal{K}},\, \mathcal{TC},\, \mathcal{T},\, \mathcal{F},\, \mathcal{P},\, \mathcal{DT})$ where

$S_{\mathcal{F}}$ is the set of function symbols,
$S_{\mathcal{P}}$ is the set of predicate symbols,
$S_{\mathcal{T}}$ is the set of type constructor symbols and
$S_{\mathcal{K}}$ is the set of type class symbols,
$\mathcal{TC}$ is the set of subclass declarations,
$\mathcal{T}$ is the set of type declarations,
$\mathcal{F}$ is the set of function declarations and
$\mathcal{P}$ is the set of predicate declarations.

They are defined just like for the polymorphic first-order language with type classes (Sect. 3.1.1). And where

$\mathcal{DT}$ is the set of data type declarations. Each element $(\tau, f_1[\tau_{1_1}, \ldots, \tau_{m_1}], \ldots, f_n[\tau_{1_n}, \ldots, \tau_{m_n}]) \in \mathcal{DT}$ is a data type declaration of the type term τ. Each such type term τ is called a *data type*, which is constructed by the function symbols $f_1, \ldots, f_n$, which are called *constructors*. For each f_i it must hold that the type variables present in the type terms $\tau_1, \ldots, \tau_{m_i}$ are also present in τ. Furthermore, for each f_i it must hold that $f_i : \forall_\tau\, \alpha_1, \ldots, \alpha_{m_i}.\, \tau'_1, \ldots, \tau'_m \to \tau_{f_i} \in \mathcal{F}$ and that $\tau_{f_i}\sigma_i$ is equal to τ if σ_i is $\{\alpha_1 \mapsto \tau_{1_i}, \ldots, \alpha_{m_i} \mapsto \tau_{m_i}\}$. A constructor must not be unifiable with other constructors (of any data type).

A data-type's type-symbol must still be declared in $\mathcal{S}_{\mathcal{T}}$ and its type declaration in $\mathcal{T}$. A constructor is *recursive* if it has an argument that is of the data type (or an instance of that data type) the constructor belongs to. A set of data types is *mutually recursive* if each data type has for each other data type at least one constructors with arguments of that data type.

Semantics

Definition 4.2.1.2 (**Structure**). Given a signature Σ for polymorphic first-order language with type classes and data types the corresponding Σ-structure is a tuple $\Sigma = (\mathcal{U}, \mathcal{D}, \mathcal{I}_{\mathcal{T}}, \mathcal{I}_{\mathcal{F}}, \mathcal{I}_{\mathcal{P}})$ where

$\mathcal{U}$ is a non-empty countable set of elements, the universe.
$\mathcal{D}$ a non-empty set of non-empty disjoint subsets of $\mathcal{U}$. It represents the set of the types (domains). For each type we require that it is non-empty.
$\mathcal{I}_{\mathcal{T}}$ is the set of type constructors ($\kappa^{\mathcal{I}}$) which map domains to a domain ($\mathcal{D}^n \to \mathcal{D}$).
$\mathcal{I}_{\mathcal{F}}$ is the set of functions ($f^{\mathcal{I}}$) which maps a cross product of domains and elements to an element ($\mathcal{D}^n \times \mathcal{U}^m \to \mathcal{U}$).
$\mathcal{I}_{\mathcal{P}}$ is the set of predicates ($p^{\mathcal{I}}$) which map a cross product of domains and elements to true or false ($\mathcal{D}^n \times \mathcal{U}^m \to \{0, 1\}$).

This is the same as the structure for the polymorphic first-order language with type classes (Sect. 3.1.4) with additional requirement on $\mathcal{D}$ and $\mathcal{I}_{\mathcal{F}}$ (Def. 4.2.1.3). I.e. the semantics of types has to be adapted so that a data type's interpretation is generated by its constructors.

A data type is a type with an interpretation that is generated by its constructors. The simplest case is a data type without any constructors that have non data type arguments. One such example are the natural numbers ($\tau_{\mathbb{N}}$), constructed by $\mathsf{0}$ and the successor function s. In the simplest case, the simple restriction on the interpretation of the data type ($\mathcal{I}(\tau_{\mathbb{N}})$) is that, considering only (term-)ground terms constructed from the constructors ($\mathsf{0}$ and s), every element ($d \in \mathcal{I}(\tau_{\mathbb{N}})$) of the interpretation of the data type must be the interpretation of exactly one such (term-)ground term.

In our setting, constructors with non data type arguments are allowed. Otherwise list, constructed by *nil* and *cons*, would not be possible, because one argument of cons is not necessarily a data type. This complicates the restriction, because multiple (term-)ground terms of a non data type can have the same interpretation. Thus, the simple restriction is not possible and a restriction that takes the interpretation of the non data type arguments into account is necessary.

Our definition that extends to data types with constructors that have non data type arguments is given by the following property that must hold for every interpretation:

Definition 4.2.1.3 (Semantic Restriction of Data Types). For each data type declaration $(\tau,\, f_1[\tau_{1_1},\, \dots,\, \tau_{m_1}],\, \dots,\, f_n[\tau_{1_n},\, \dots,\, \tau_{m_n}]) \in \mathcal{DT}$, for each instance $\tau\theta$ of τ and for each $d \in \mathcal{I}(\tau\theta)$ there exists at least one (term-)ground term t of type $\tau\theta$ such that $\mathcal{I}(t) = d$. Furthermore, for all (term-)ground terms $t = f\langle\tau_1,\, \dots,\, \tau_n\rangle(t_1,\, \dots,\, t_m)$ and $s = f'\langle\tau'_1,\, \dots,\, \tau'_{n'}\rangle(s_1,\, \dots,\, s_{m'})$ such that t, s are of type $\tau\theta$, $\mathcal{I}(t) = d$ and $\mathcal{I}(s) = d$, it holds that $f = f'$, that f is a constructor of the data type declaration of τ, that $\mathcal{I}(\tau_1) = \mathcal{I}(\tau'_1),\, \dots,\, \mathcal{I}(\tau_n) = \mathcal{I}(\tau'_n)$ and that $\mathcal{I}(t_1) = \mathcal{I}(s_1),\, \dots,\, \mathcal{I}(t_m) = \mathcal{I}(s_m)$.

The Semantic Restriction of Data Types Is Not Finitely Expressible in First-Order

This restriction cannot be expressed by a finite first-order axiomatization. Its goal is to uniquely characterize the interpretation of the data type to exactly the set that is generated by the data type's constructor terms such that the constructor terms are distinct. The distinct part can be axiomatized in first-order logic:

Definition 4.2.1.4 (Distinct). Terms with different constructors as top symbols are not equal:

$$f_i\langle\tau_1,\, \dots,\, \tau_{m_1}\rangle(t_1,\, \dots,\, t_{n_1}) \not\approx f_j\langle\tau'_1,\, \dots,\, \tau'_{m_2}\rangle(s_1,\, \dots,\, s_{n_2}) \text{ if } i \neq j$$

Definition 4.2.1.5 (Injective). Terms with the same constructor top symbol and type arguments are only equal if all their term arguments are equal:

$$f_i\langle\tau_1,\, \dots,\, \tau_m\rangle(t_1,\, \dots,\, t_n) \approx f_i\langle\tau_1,\, \dots,\, \tau_m\rangle(s_1,\, \dots,\, s_n) \implies (s_1 \approx t_1 \wedge \ldots \wedge s_n \approx t_n)$$

The 'no-junk' part, i.e. that there is no element of the type's interpretation that does not have a corresponding constructor term, cannot be finitely axiomatized in first-order logic. It follows from the Induction principle:

Definition 4.2.1.6 (Induction). A property (over a data type) holds if for all elements e (of that data type) the assumption that the property holds for all elements that are smaller (according to the well-founded relation, i.e. for structural induction the proper subterm relation) then e implies that the property holds for e:

$$\mathcal{I}(\forall X : \tau.\, \phi) \text{ if}$$

$$\text{for all } d \in \mathcal{I}(\tau) \text{ it holds that for all } e \in \mathcal{I}^{\prec_d}(\tau), \mathcal{I}_{\{X \mapsto e\}}(\phi) \text{ implies } \mathcal{I}_{\{X \mapsto d\}}(\phi)$$

and τ is a data type, t_d is the constructor term such that $d = \mathcal{I}(t_d)$ and $\mathcal{I}^{\prec_d}(\tau)$ is the set of all interpretations of proper constructor subterms of t_d

The (semantic) induction principle can also be expressed by an induction schema, which can be axiomatized in first-order logic by countably many axioms. This is captured by the following schema:

Definition 4.2.1.7 (**Induction Schema).** Let ϕ be an arbitrary first-order formula and X be a variable such that its type τ is a data type and such that X occurs in ϕ. Then the following is the induction schema for data types in first-order logic:

$$(Case_1 \wedge \cdots \wedge Case_n) \implies \phi[X]$$

where there is an induction $Case_i$ for each constructor f_i of τ defined as:

$$\forall X_1, \ldots, X_m. \, (IH_1 \wedge \cdots \wedge IH_m) \implies \phi[f_i(X_1, \ldots, X_m)]$$

where each X_i is a fresh (schema) variable and all IH_i are omitted (i.e. $\top$) if the type of the corresponding X_i is not τ. If the type of the corresponding X_i is τ, then IH_i is defined as:

forall terms X_i' (of type τ) smaller or equal (by the subterm relation) than X_i it holds that $\phi[X_i']$

Because of the refutational setting, when we turn the induction schema into an SupInd inference, it is applied to Skolem functions. In SupInd, the individual cases of induction are therefore negated and then refuted: I.e. the $Case_i$ turn into showing unsatisfiablity (together with the axioms and previous induction hypotheses) of

$$\neg(\forall X_1, \ldots, X_m. \, (IH_1 \wedge \cdots \wedge IH_m) \implies \phi[f_i(X_1, \ldots, X_m)])$$

respectively, after Skolemization (i.e. replacing each variable X_i by its Skolemization sk_i), showing unsatisfiablity (together with the axioms and previous induction hypotheses) of

$$IH_1 \wedge \cdots \wedge IH_m \wedge \neg\phi[f_i(\mathsf{sk}_1, \ldots, \mathsf{sk}_m)]$$

4.2.2. The SupInd Calculus

Proving a (first-order) conjecture is equivalent to showing its negation to be unsatisfiable. Superposition requires that the axioms and the negated conjecture are Skolemized and clausified. In the refutational setting, universal quantification of a conjecture turns into an existential one in the negated conjecture. The existential quantification is then represented by Skolem functions.

One of the main strengths of superposition is redundancy elimination. Superposition can remove clauses if they are reducible to smaller ones. A set of clauses N is *reducible* to a set of clauses N' if N is entailed by N' and all clause in N' are smaller than the ones in N (according to a clause and a term ordering). The smaller clauses could, for example, be introduced by rewriting or by inferences. Superposition supports any simplification that can be phrased in terms of redundancy elimination. Superposition (and in turn SupInd) perform simplification during the proof search. To profit from simplifications performed on the conjecture, the induction cases and their hypothesis are directly generated from the (simplified) negated conjecture's clauses, instead of using the initial conjecture's clauses. The evaluation (Sect. 4.7) shows that Pirate can solve 6 more problems (+9%) than the next closed tool (HipSpec), when using no strengthening techniques. I believe that the use of the simplified negated conjecture is the main reason for this.

Therefore, the calculus must keep track of the (negated) conjecture and induction hypotheses by splitting the clauses superposition operates on into three parts: the (negated) conjecture clauses (N_C), the induction hypotheses clauses (N_{IH}), and the axioms and all inference result clauses (N_A). The SupInd calculus' inferences rules work on states. A triple $(N_A,\, N_{IH},\, N_C)$ corresponds to a simple state of our calculus. A simple state is unsatisfiable if one of its clause sets contains the empty clause. Complex states represent a split into two cases. A complex state is unsatisfiable if either both substates are unsatisfiable (&) or if one of the two substates (|) is unsatisfiable. Complex states are introduced by induction steps. The initial state is the simple state $(N_A,\, \emptyset,\, N_C)$ where N_A are the initial axiom clauses and the N_C the initial negated conjecture clauses.

$$\mathit{state} ::= \;(N_A,\, N_{IH},\, N_C) \;\;\Big|\;\; \mathit{state}\,\&\,\mathit{state} \;\;\Big|\;\; \mathit{state}\,|\,\mathit{state}$$

A simple state $(N_A,\, N_{IH},\, N_C)$ is a *first-order consequence* of another simple state $(N'_A,\, N'_{IH},\, N'_C)$ if $N_A \cup N_{IH} \cup N_C$ is a first-order consequence of $N'_A \cup N'_{IH} \cup N'_C$. A simple state $(N_A,\, N_{IH},\, N_C)$ is *entailed* (*implied*) by a simple state $(N'_A,\, N'_{IH},\, N'_C)$ if $N_A \cup N_{IH} \cup N_C$ is entailed by $N'_A \cup N'_{IH} \cup N'_C$. I use the symbol & for the states where both substates must be unsatisfiable for the &-state to be unsatisfiable, because that state represents the conjunction of the conjectures of the two substates (i.e. both cases have to be refuted/proven). Similarly, I use the symbol | for the states where one of the two substates must be unsatisfiable, because that state represents the disjunction of the conjectures of the two substates (i.e. only one case has to be refuted/proven).

The SupInd calculus consists of inference rules that can be applied to any state or substate which fulfills their premise and side conditions, creating a new state which replaces the premises by the inference rule's conclusion. We will now formally define all inference rules and their side conditions followed by a short description of the inference rules. The rules themselves are presented as inferences, where the upper part is the premise (sub)state and the lower part is a state that replaces the premise.

Superposition Let $n \in \{1,2\}$ and for all $i \leq n$ let $C_i \in N_A \cup N_{IH} \cup N_C$.

$$\frac{(N_A,\, N_{IH},\, N_C)}{(N_A \cup \{C'\},\, N_{IH},\, N_C)}\ \textsc{Inf} \quad \text{if} \quad \frac{C_1 \quad \cdots \quad C_n}{C'} \quad \text{is a superposition inference}$$

Redundancy Elimination
Let N, N' be sets of clauses such that N is reducible to N' with respect to $N_A \cup N_{IH} \cup N_C$.

$$\frac{(N_A \uplus N,\, N_{IH},\, N_C)}{(N_A \cup N',\, N_{IH},\, N_C)}\ \textsc{Red}_A \qquad \frac{(N_A,\, N_{IH} \uplus N,\, N_C)}{(N_A,\, N_{IH} \cup N',\, N_C)}\ \textsc{Red}_{IH} \qquad \frac{(N_A,\, N_{IH},\, N_C \uplus N)}{(N_A,\, N_{IH},\, N_C \cup N')}\ \textsc{Red}_C$$

Derive Facts Let N be an arbitrary set of clauses, let $cnf(\neg N)$ be the clause normal form of the negation of N and let SupInd* be any finite chain of applications of SupInd's rules.

$$\frac{(N_A,\, N_{IH},\, N_C)}{(N_A,\, N_{IH},\, N_C) \mid (N_A \cup N,\, N_{IH},\, N_C)}\ \textsc{Derive} \quad \text{if} \quad N \text{ is a consequence of } N_A \cup N_{IH} \cup N_C$$

Unsatisfiability Propagation

Let S be an arbitrary state and let $\bot \in N_{A_l} \cup N_{IH_l} \cup N_{C_l}$ and $\bot \in N_{A_r} \cup N_{IH_r} \cup N_{C_r}$.

$$\frac{(N_{A_l},\, N_{IH_l},\, N_{C_l})\ \&\ (N_{A_r},\, N_{IH_r},\, N_{C_r})}{(\{\bot\},\, \{\bot\},\, \{\bot\})}\ \& \qquad \frac{(N_{A_l},\, N_{IH_l},\, N_{IH_l}) \mid S}{(\{\bot\},\, \{\bot\},\, \{\bot\})}\ |_{\mathrm{l}} \quad \frac{S \mid (N_{A_r},\, N_{IH_r},\, N_{IH_r})}{(\{\bot\},\, \{\bot\},\, \{\bot\})}\ |_{\mathrm{r}}$$

Structural Induction

Let sk be occurring in N_C, let sk be a Skolem constant of a data type and let $f_1, \ldots, f_n$ be the data type's constructors.

$$\frac{(N_A,\, N_{IH},\, N_C)}{(N_A,\, N_{IH},\, N_C) \mid (Case_{f_1}\ \&\ \cdots\ \&\ Case_{f_n})}\ \textsc{Ind}$$

where $Case_{f_i}$ is defined as

$$(N_A,\, N_{IH} \cup hyps_{f_i},\, N_C)[\mathsf{sk} \mapsto f_i(\mathsf{sk}_1,\, \ldots,\, \mathsf{sk}_m)]$$

$\mathsf{sk}_1, \ldots, \mathsf{sk}_m$ are fresh Skolem constants and the new induction hypotheses ($hyps_{f_i}$) are defined as

$$\bigcup_{\mathsf{sk_j} \text{ is of } \mathsf{sk}\text{'s type}} (cnf(\neg N_C)\ \cup\ N_{IH})[\mathsf{sk} \mapsto \mathsf{sk_j}]$$

where $cnf(\neg N_C)$ is the clause normal form of the negation of N_C (the negated conjecture), where the implicit universal quantifiers are placed directly below the negation and where sk must be treated as a constant, i.e. not be replaced by a variable. The other Skolems can be treated as implicit existential quantifications directly below the negation. In Pirate there is an option for both behaviors.

For a simpler presentation, the Structural Induction rule is defined for Skolem constants. To generalize it to Skolem functions, we must add the arguments of the initial Skolem function (sk) to each freshly created Skolem function.

The inferences of the standard superposition calculus are lifted by SupInd's Superposition rule to be inferences on simple states in such a way that inference results are always added to the axiom set N_A. Superposition's redundancy is lifted by the Redundancy Elimination rule such that simplified clauses are added to the set their main premise came from.

With induction it is often necessary to guess and prove auxiliary lemmas that would not be derived by first-order inferences, i.e. lemmas with proofs that require induction. Therefore, we allow to add arbitrary clause sets with the Derive Facts rule if they can be shown to follow (by possibly nested induction and first-order reasoning) from the axioms, induction hypotheses and conjecture. The Propagation rules simplifies complex states by collapsing complex states where the premises already have the required unsatisfiablity results. Finally the Structural Induction rules generates the appropriate induction cases with the induction hypotheses, that would otherwise require an induction schema. Because the conjecture was negated and Skolemized, the Structural Induction rule applies to Skolem functions instead of universally quantified variables. Induction preserves the premise's simple state in order to guarantee refutational completeness of each simple state for the purely first-order fragment (i.e. without data-type declarations). In the presence of nested inductions it is not obvious if it is possible to (efficiently) preserve refutational completeness for the purely first-order fragment without preserving the premise's simple state.

Soundness

The rules of the SupInd calculus are sound.

Lemma 4.2.2.1 (The Superposition Rule is Sound)**.**
The result of the Superposition rule is implied by its premise.

Proof. The superposition calculus is sound, therefore the inferred clause (C') is a consequence of the premises (C_1 and C_2) of the superposition inference. Since the premises (C_1 and C_2) are also part of the simple state premise ((N_A, N_{IH}, N_C)) of the Superposition rule, the resulting simple state (($N_A \cup \{C'\}$, N_{IH}, N_C)) is a first order consequence of the premise. □

Lemma 4.2.2.2 (The Redundancy Elimination Rules are Sound)**.**
The result of the Redundancy Elimination rules are implied by their premise.

Proof. The Redundancy Elimination rule RED_A replace a subset N of the axiom clause set (N_A) by a clause set N' such that N' implies N (with respect to the remaining clauses of the simple set, $N_A \cup N_{IH} \cup N_C$). Since the clause set N' implies the clause set N, the clause set $N_A \cup N'$ implies the clause set $N_A \cup N$. Thus, the resulting simple state of the inference rule is a first-order consequence of the premise.

The proof for the two other Redundancy Elimination rules is analogous. □

Lemma 4.2.2.3 (The Derive Facts Rule is Sound)**.**
The result of the Derive Facts rule is implied by its premise.

Proof. By the definition of the Derive Facts rule, the clause set N is a consequence of the clause set $N_A \cup N_{IH} \cup N_C$. Thus, $N_A \cup N_{IH} \cup N_C$ is satisfiable if and only if $N_A \cup N \cup N_{IH} \cup N_C$ is satisfiable. Therefore, the simple state's axiom set can be extended by N to the resulting simple state ($N_A \cup N$, N_{IH}, N_C). □

Lemma 4.2.2.4 (The Unsatisfiability Propagation Rule for & is Sound)**.**
The result of the Unsatisfiability Propagation rule for & is implied by its premises.

Proof. By definition of the &-state, it is unsatisfiable if both its premises are unsatisfiable. If the empty clause is contained in one of the clause sets of a simple state, that simple state is unsatisfiable. Therefore, if both premise simple state contain the empty clause the &-state can be simplified to an unsatisfiable simple state. □

Lemma 4.2.2.5 (The Unsatisfiability Propagation Rules for $|$ are Sound)**.**
The result of the Unsatisfiability Propagation rules for $|$ is implied by its premises.

Proof. By definition of the $|$-state, it is unsatisfiable if at least one of its premises is unsatisfiable. If the empty clause is contained in one of the clause sets of a simple state, that simple state is unsatisfiable. Therefore, if the left or the right premise is a simple state that contains the empty clause, then the $|$-state can be simplified to an unsatisfiable simple state. $\square$

Lemma 4.2.2.6 (The Structural Induction Rule is Sound)**.**
The result of the Structural Induction Rule is implied by its premise.

Proof. A $|$-state is implied by a simple state if both its left and right hand side are implied by the simple state.

The simple state on the left-hand side of the resulting $|$-state is the unchanged premise and thus implied by the premise.

The induction principle justifies the splitting of the conjecture in a case ($Case_i$) for each data-type constructor (f_i). All cases have to be shown, i.e. for each case the negated conjecture has to be shown unsatisfiable. Thus, the replacement of sk by $f_i(sk_1, \dots, sk_m)$ in the creation of the cases $((N_A, N_{IH} \cup hyps_{f_i}, N_C)[sk \mapsto f_i(sk_1, \dots, sk_m)])$ is justified by the induction principle (and the negation of the conjecture). The previous induction hypotheses can be treated as axioms, and all axioms can be kept. Thus, it remains to show that the newly introduced induction hypotheses ($hyps_{f_i}$) are justified.

The induction principle justifies an induction hypothesis for each argument of the same type. Therefore, for each argument sk_j of $f_i(sk_1, \dots, sk_m)$ that is of the same type as $f_i(sk_1, \dots, sk_m)$ and an induction hypothesis is added. Since N_C is the negated conjecture, to derive the induction hypothesis N_C is negated. This justifies the $\bigcup_{sk_j \text{ is of } sk\text{'s type}} (cnf(\neg N_C))[sk \mapsto sk_j]$ part of the induction hypothesis. Because I use strong induction, the induction hypothesis hold for all structurally smaller terms. Therefore, the previous induction hypotheses also hold for all terms structurally smaller terms (than sk), i.e. all arguments of f_i. So while the induction hypothesis clauses for the sk_j are not yet present, they are justified by previous induction steps. Thus, the $\bigcup_{sk_j \text{ is of } sk\text{'s type}} N_{IH}[sk \mapsto sk_j]$ part of the induction is permissible. $\square$

Refutational Completeness for the First-Order Fragment

Assuming fairness (as defined below), SupInd guarantees that for any simple state that is first-order unsatisfiable it will eventually derive the empty clause.

The *purely first-order* fragment of polymorphic first-order language with type classes and data types is the part without the data-type declarations ($\mathcal{DT}$). A *run* of the SupInd calculus is a sequence of states $N_0 \vdash N_1 \vdash N_2 \vdash \cdots$ such that N_{i+1} follows by an application of a SupInd rule to N_i. Let $\succ$ be a well-founded ordering on states. A state N is *redundant with respect to* a state N' if $N \succ N'$ and there exists a run such that $N \vdash \cdots \vdash N'$. A SupInd rule is *enabled* with respect to a state N if all its premises are contained in N and its result is not redundant. A run is *fair* if for each i and SupInd rule *inf* that is enabled at N_i, there exists a j with $j > i$ such that *inf* is not enabled in N_j.

Theorem 4.2.2.7 (Dynamic Refutational Completeness for Purely First-Order Logic)**.**
Let $N_0 \vdash N_1 \vdash N_2 \vdash \cdots$ be a SupInd run that is fair for SupInd's Superposition rule and N_* be its limit. Then each simple state $(N_A,\ N_{IH},\ N_C)$ in N_* has a purely first-order model if and only if it does not contain the empty clause, i.e. $\bot \notin N_A \cup N_{IH} \cup N_C$.

Proof. The goal is to show that a SupInd run that is fair for SupInd's Superposition rule is a fair run for superposition on the union of each simple state's clause sets.

The Derive Facts and the Structural Induction inference rules of SupInd preserve a copy of the simple state that they are applied to. Thus, they do not interfere with first-order reasoning, because it can continue on the preserved copy.

The Unsatisfiability Propagation inference rule removes states (and their simple substates) when they are no longer required for the overall proof. Since the states are removed they cannot be part of N_*.

The Redundancy Elimination rules, perform only changes in the clause sets of the simple state that would also be allowed in superposition on the union of the clause sets. The Superposition rule applies all inferences of the superposition calculus, that the dynamic refutational completeness for polymorphic first-order language with type classes requires for the union of the clause sets of the simple state (Theorem 3.5.4.11). Thus, purely first-order dynamic refutational completeness for the simple state holds. □

Refutational Completeness

Assuming fairness (as defined below) SupInd guarantees that for any simple state that is unsatisfiable when combined with the countable set of instances of the structural induction schema, it will eventually derive the empty clause.

The induction principle of a data type cannot be expressed in first-order logic, because it is quantified over all properties. Therefore, it has to be an inference rule in the SupInd calculus, instead of being stated as an axiom. But for a fixed first-order property, the induction principle can be stated in first-order logic. As explained before, this is done with the help of an induction schema (Def. 4.2.1.7). Furthermore, a given (countable) first-order signature only allows countably many properties, i.e. it is possible to enumerate all possible properties (and their induction principles). They are enumerable, because they have a syntactic representation as first-order formulas and that representation is enumerable.

Let N_{Ind} be the set of clauses resulting from all such properties, i.e. the clauses resulting from all instances of the induction schema for all possible properties.

Static refutational completeness of superposition (saturated sets without $\bot$ have a model, Lemma 3.5.4.8) and compactness of first-order logic (proofs are finite, Lemma 3.5.4.10) extend to countable signatures and countable clause sets. In practice, static refutational completeness of infinite saturated sets is not useful, because we can not handle infinite sets of clauses in a solver. By integrating induction into SupInd's calculus, the countable set of instances of the induction schema can be finitely represented and thus be handled in practice. I call a clause set N *inductively unsatisfiable* if $N \cup N_{Ind}$ is unsatisfiable. Dynamic refutational completeness for SupInd holds, i.e. for any inductively unsatisfiable simple state there is a sequence of SupInd inference steps that leads to the simple state being refuted.

Theorem 4.2.2.8 (Dynamic Refutational Completeness).
Let N_{Ind} be the clause set corresponding to the countable set of (first-order) structural induction principles for the declared data types. Let N_0 be a simple state containing only finitely many clauses and $N_0 \vdash N_1 \vdash N_2 \vdash \cdots$ be a SupInd run that is fair for all of SupInd rules (except possibly Redundancy Elimination) and let N_* be its limit. Then $N_{Ind} \cup N_0$ has a model if and only if N_* is not a simple state containing the empty clause.

Proof. If N_* does not have a model (i.e. is a simple state containing the empty clause), then obviously $N_{Ind} \cup N_0$ cannot have a model. If $N_{Ind} \cup N_0$ does not have a model, then compactness ensures that there is a finite superposition proof using finitely many clauses from N_{Ind}. Those clauses are present in N_*, because a fair run of SupInd introduces them eventually. Suppose they are not, then the run would not be fair, because the corresponding Derive Facts rule(s) would have to be enabled forever (because N_* is the limit). Thus, N_* contains the empty clause if $N_{Ind} \cup N_0$ does not have a model. □

4.3. A Strategy for Applying Structural Induction

As we have seen, SupInd is refutationally complete with respect to the countable set of first-order instances of the induction schema, i.e. for a conjecture N such that $N \cup N_{Ind}$ is unsatisfiable SupInd will eventually derive the empty clause. In practice, refutational completeness must be augmented by heuristics that guide the proof search into practically relevant parts. Otherwise prolific inferences, e.g. the Derive Facts rule, make exploring the search space unrealistic and thus finding a refutation unfeasible. In the this section I describe a heuristic to determine how to automatically chose the next Skolem function, of a simple state, to apply induction on (Sect. 4.3.1) and a technique to recover from choosing Skolem functions that turn out to be irrelevant for the current partial proof (Sect. 4.3.2).

The restriction of the induction inference to Skolem functions is possible, because induction over arbitrary terms is covered by the Derive Facts inference introducing the corresponding lemmas. Since many inductive proofs require additional lemmas, techniques to determine the lemmas which should be derived next are especially important and are described in the next section (Sect. 4.4).

4.3.1. Which Skolem Function to Apply Induction On

One of the main challenges of inference-based automated theorem proving is to choose the next inference to perform. Always picking the right inference would eliminate the need to explore most of the search space. This is even more so a challenge for automating induction with prolific inference rules like the Structural Induction rule and the Derive Facts rule. In particular, we would like to automatically determine where to apply the induction inference next.

Let us return to the associativity property of addition $(+)$ on the natural numbers $(\mathbb{N})$, constructed by $\mathit{0}$ and the successor function s and the initial state $(N_A,\ N_{IH},\ N_C)$ where

$$N_A = \{\mathit{0} + Y \approx Y,\ \mathsf{s}(X) + Y \approx \mathsf{s}(X+Y)\} \qquad N_{IH} = \emptyset \qquad N_C = \{\mathsf{x} + (\mathsf{y} + \mathsf{z}) \not\approx (\mathsf{x} + \mathsf{y}) + \mathsf{z}\}$$

and addition is recursively defined by N_A. In the introduction (Sect. 4.1), we observed that selecting induction over y and z requires further inductive reasoning, but induction over x immediately yields first-order refutations. Nonetheless, it is hard to come up with a comprehensive set of rules that capture which Skolems are preferable for all possible cases, e.g. that some recursively defined argument positions are preferable. This is especially difficult for more complex settings with many function symbols and many axioms.

Instead of a complex and fixed rule set, we use the insight that, in practice, base cases (e.g. for $\mathbb{N}$: *0*) often require only purely first-order reasoning. In particular, no nested inductions are possible in the base cases, if the base-case constructors' arguments are not of a data type. While auxiliary lemmas might still be necessary (Sect. 4.4), this section focuses on applying the Structural Induction rule. For example, in the case of *nil* and *0* no nested inductions are possible, but an *nat-pair* data type would have an argument of data type (*nat*) for its (base-case) constructor. Furthermore, finding a proof for the base cases is an obvious prerequisite to proving all induction cases.

To use this idea, one could try all base cases of all Skolem functions and then choose one of the Skolem functions that has first-order proofs for (most of) its base cases. Unfortunately, this approach requires all base cases for all Skolem functions to be tried. In addition, no information can be gained if (nested) inductions on multiple Skolem functions are necessary.

To solve this issue, I create an intermediate clause set, which simultaneously over-approximates all base cases of all Skolem functions of the original simple state. In particular, if one combination of base cases has a proof, then so does the over-approximated clause set. Refuting the over-approximated clause set gives us some base cases that are provable. Thus, an induction on those Skolem functions is (at least for the base cases) relevant for the proof. Creating an over-approximation enables testing the suitability of all Skolem functions occurring in the conjecture by a single proof attempt.

Creating an Over-Approximation of the Base Cases To create the over-approximated clause set, all Skolem functions are treated as if they were base-case constructors. This allows the heuristic to be applied also in cases with multiple base-case constructors. The starting point of the intermediate clause set is the union of the clause sets (N_A, N_{IH}, N_C) of the simple state. This clause set is then extended by adding a new axiom for each triple of an original axiom (i.e. each clause occurring in N_A), a Skolem function (occurring in the negated conjecture N_C) and a base case constructor (occurring in the axiom). The new axiom is created by replacing the base-case constructor terms occurring in the original axiom by the Skolem functions (of the same type).

Replacing the Skolem functions by the base case constructors (and not vice-versa) enables the use of this heuristic also in the presence of multiple base-case constructors. For example, from the original axioms (1) $0 + Y \approx Y$ and (2) $s(X) + Y \approx s(X + Y)$ only (1) contains a base-case constructor (i.e. the constructor *0*). Axiom clause (1) is then over-approximated for the base-cases by adding the new clause (3) for the combination with *0* and x, the new clause (4) for the combination with *0* and y and the new clause (5) for the combination with *0* and z. This results in the following clause set (N_A):

$$\begin{array}{llcl}
(1) & \mathit{0}+Y & \approx & Y \\
(2) & s(X)+Y & \approx & s(X+Y) \\
(N_C) & \mathsf{x}+(\mathsf{y}+\mathsf{z}) & \not\approx & (\mathsf{x}+\mathsf{y})+\mathsf{z}
\end{array}
\qquad\qquad
\begin{array}{ll}
(3) & \mathsf{x}+Y\approx Y \\
(4) & \mathsf{y}+Y\approx Y \\
(5) & \mathsf{z}+Y\approx Y
\end{array}$$

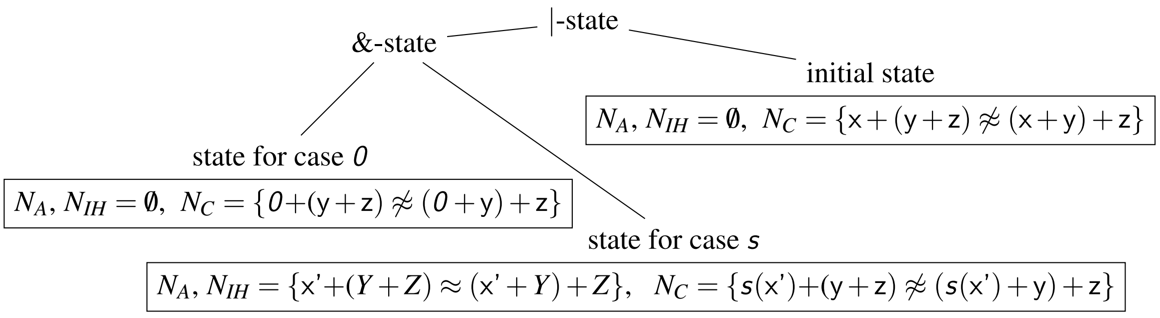
If the clause set is shown unsatisfiable, we analyze the resulting proof to find out which Skolem functions are preferable. The generated clause set can allow multiple different proofs, which possibly use different axioms or have different dependencies. After finding the first proof, it is possible to continue the proof search to get alternative proofs (e.g. up to a time limit). In Pirate I only consider the first proof, to avoid wasting time that might be better spent elsewhere.

Let us consider an example proof of the negated conjecture using the generated clauses:

(N_C)	$\mathsf{x}+(\mathsf{y}+\mathsf{z})\not\approx(\mathsf{x}+\mathsf{y})+\mathsf{z}$	The Negated Conjecture
(a)	$\mathsf{x}+\mathsf{z}\not\approx(\mathsf{x}+\mathsf{y})+\mathsf{z}$	using (4) and (N_C)
(b)	$\mathsf{z}\not\approx(\mathsf{x}+\mathsf{y})+\mathsf{z}$	using (3) and (a)
(c)	$\mathsf{z}\not\approx\mathsf{y}+\mathsf{z}$	using (3) and (b)
(d)	$\mathsf{z}\not\approx\mathsf{z}$	using (4) and (c)
(e)	$\perp$	using (d)

A first-order reasoning step (e.g. (b)) *depends* on another step (e.g. (a)) if step (b) uses a clause whose derivation involves step (a).

The new axiom (5) for z is not used in the proof and hence z is not considered for induction. The axioms (3) and (4) for x and y are both used in the proof. We consider the dependencies between the steps and prefer those Skolem functions where the application of the respective clause does not depend on another clause of a different Skolem function. All inferences belonging to y depend on the inferences of x. Thus, x is selected for induction. In case we have multiple preferred Skolem functions, I try to determine if one of those functions alone is sufficient for the (base-case) proof, by running another proof attempt which excludes the axioms of the other of the possibly-preferred Skolem functions. One Induction step on x yields a proof. Here we show the corresponding state (omitting the content of the N_A set, which is the same for all simple states):


|-state
&-state
initial state
$N_A, N_{IH}=\emptyset,\ N_C=\{\mathsf{x}+(\mathsf{y}+\mathsf{z})\not\approx(\mathsf{x}+\mathsf{y})+\mathsf{z}\}$
state for case $\mathit{0}$
$N_A, N_{IH}=\emptyset,\ N_C=\{\mathit{0}+(\mathsf{y}+\mathsf{z})\not\approx(\mathit{0}+\mathsf{y})+\mathsf{z}\}$
state for case s
$N_A, N_{IH}=\{\mathsf{x'}+(Y+Z)\approx(\mathsf{x'}+Y)+Z\},\ \ N_C=\{s(\mathsf{x'})+(\mathsf{y}+\mathsf{z})\not\approx(s(\mathsf{x'})+\mathsf{y})+\mathsf{z}\}$

The first-order refutations for the base case ($\mathit{0}$) and the step case (s) are as follows:

($\mathit{0}$a)	$\mathit{0}+(\mathsf{y}+\mathsf{z}) \not\approx (\mathit{0}+\mathsf{y})+\mathsf{z}$	
($\mathit{0}$b)	$\mathsf{y}+\mathsf{z} \not\approx (\mathit{0}+\mathsf{y})+\mathsf{z}$	by (2)
($\mathit{0}$c)	$\mathsf{y}+\mathsf{z} \not\approx \mathsf{y}+\mathsf{z}$	by (2)
($\mathit{0}$d)	$\perp$	

(sa)	$\mathit{s}(\mathsf{x'})+(\mathsf{y}+\mathsf{z}) \not\approx (\mathit{s}(\mathsf{x'})+\mathsf{y})+\mathsf{z}$	
(sb)	$\mathit{s}(\mathsf{x'}+(\mathsf{y}+\mathsf{z})) \not\approx (\mathit{s}(\mathsf{x'})+\mathsf{y})+\mathsf{z}$	by (1)
(sc)	$\mathit{s}(\mathsf{x'}+(\mathsf{y}+\mathsf{z})) \not\approx (\mathit{s}(\mathsf{x'}+\mathsf{y}))+\mathsf{z}$	by (1)
(sd)	$\mathit{s}(\mathsf{x'}+(\mathsf{y}+\mathsf{z})) \not\approx \mathit{s}((\mathsf{x'}+\mathsf{y})+\mathsf{z})$	by (1)
(se)	$\mathit{s}((\mathsf{x'}+\mathsf{y})+\mathsf{z}) \not\approx \mathit{s}((\mathsf{x'}+\mathsf{y})+\mathsf{z})$	by N_{IH}
(sf)	$\perp$	

In Pirate we always use this heuristic to determine the next Skolem function. In the case of a time out we randomly select a Skolem function.

Guarantees of the Heuristic In general, we have to apply superposition on the generated clause set with a (time) limit, because we cannot guarantee that the clause set is first-order unsatisfiable. Nonetheless, if one of the base cases is first-order unsatisfiable, then so is the generated clause set. Furthermore, if the clause set is first-order satisfiable then we know that the (nested) induction's base cases are first-order satisfiable, too. Thus, we know that (nested) induction is futile without discovering additional lemmas.

One of the main applications of automating structural induction is proving properties of functional programs. In particular proving properties of primitive recursive functions. In essence, *primitive recursive functions* are those functions that are defined using exhaustive case distinction over data-type constructors. Furthermore, the defining equations (i.e. the axioms) are universally quantified, which implies the absence of Skolem functions. Thus, in this setting and with a suitable (term) ordering the axioms are (ground) confluent.

If the axioms are (ground) confluent and if we further assume that no base-case constructor has an argument of a data type, then superposition on the clause set generated by the heuristics terminates. This holds even if each base case on its own is satisfiable, i.e. even if all base cases would requires further nested inductions. Moreover, in this particular case the generated clause set is satisfiable only if the conjecture is not a theorem. This is particularly useful, because it can be used to discard propositions for auxiliary lemmas that are not valid.

In the more general cases, i.e. in the presence of base-case constructors with data type arguments, we can guarantee that if is some combination of base cases is first-order unsatisfiable, the generated clause set is also first-order unsatisfiable.

4.3.2. Extension to Step-Case Proofs

In the presented refutational completeness proofs for SupInd, fairness requires that eventually all applications of the induction inference rule are applied to each simple state (i.e. induction on all Skolems). It is possible to restrict the Structural Induction inference rule so that it is applied only once per simple state, i.e. that it is not applied again to the unchanged simple states that are results of the induction rule (which are only kept to ensure first-order completeness).

Applying the inference only once, i.e. to one Skolem function, results in a exploration of the search space similar to depth-first search. Applying it exhaustively results in a breadth-first like exploration. Both variants are equally expressive if an additional fairness property is observed:

For all induction cases, every Skolem function is eventually selected for performing induction in some nested induction step.

I expect the heuristic presented above to generally choose good Skolem functions. Nonetheless, each Skolem function that was selected and turns out to be irrelevant causes longer proofs. If such an irrelevant Skolem function is selected for the induction inference, the proof is duplicated (and must therefore be refound) for each of its induction cases. For example, choosing z (in the proof of the associativity conjecture above) before choosing x leads to a proof that is at least twice the size, because it duplicates the successful proof by induction over x for the two cases of induction over z. For an efficient solver, we need a solution that removes or at least reduces the costs of irrelevantly selected Skolem functions. This allows us to always apply the induction rule once per simple state, reducing the search space that needs to be explored.

A solution originates from the tight integration of induction and superposition, which allows us to (immediately) inspect and analyze the (partial) proof for each proven (nested) induction case. This analysis determines which induction steps are necessary and which can be omitted. In general, the proof analysis is (substantially) more efficient than the proof search that results in the proof. The analysis works in two phases. First the proof is minimized by removing unnecessary first-order reasoning steps. An example of a common cause of superfluous first-order reasoning steps is (greedy) rewriting. Then the proof is minimized by removing unnecessary inductive reasoning steps. Let *t* be a constructor term and z be a Skolem function such that both occur in the proof and *t* was introduced as a case of the induction inference on z. The analysis checks for each such *t* if the proof is still correct if *t* is replaced by z.

Let us return to the current example conjecture, the associativity property. Assume we have already performed two inductions, first on z and then on x and we have discovered a proof for the case where $z = s(z')$ and $x = s(x')$:

$(s\text{a})$	$s(x')+(y+s(z')) \not\approx (s(x')+y)+s(z')$	
$(s\text{b})$	$s(x'+(y+s(z'))) \not\approx (s(x')+y)+s(z')$	by (1)
$(s\text{c})$	$s(x'+(y+s(z'))) \not\approx (s(x'+y))+s(z')$	by (1)
$(s\text{d})$	$s(x'+(y+s(z'))) \not\approx s((x'+y)+s(z'))$	by (1)
$(s\text{e})$	$s((x'+y)+s(z')) \not\approx s((x'+y)+s(z'))$	by N_{IH}
$(s\text{f})$	$\bot$	

Then we can check if one of the induction steps was superfluous:

Induction on z not relevant?

Replay with $s(z') \mapsto z$

$(s\text{a}')$	$s(x')+(y+z) \not\approx (s(x')+y)+z$	
$(s\text{b}')$	$s(x'+(y+z)) \not\approx (s(x')+y)+z$	by (1)
$(s\text{c}')$	$s(x'+(y+z)) \not\approx (s(x'+y))+z$	by (1)
$(s\text{d}')$	$s(x'+(y+z)) \not\approx s((x'+y)+z)$	by (1)
$(s\text{e}')$	$s((x'+y)+z) \not\approx s((x'+y)+z)$	by N_{IH}
$(s\text{f}')$	$\bot$	

Induction on x not relevant?

Replay with $s(x') \mapsto x$

$(s\text{a}')$	$x+(y+s(z')) \not\approx (x+y)+s(z')$
$(s\text{b}')$	$? \not\approx (x+y)+s(z')$

The replay of the proof is always successful if the term *t* introduced by induction (here $s(z')$) only occurs in the instantiated part of the rules used in first-order reasoning. The possibly relevant induction steps are those that do not only occur in the instantiated part and can be discovered by one iteration over the proof.

The combination of the base-case heuristic and the step-case proof analysis allows us to guide the proof search to those Skolem Functions that have more useful base cases and efficiently recover from situations where futile induction steps were performed.

4.4. Conjecture Strengthening

A further challenge for automated induction is that many valid conjectures cannot be proven directly by induction, but they follow from lemmas that are provable by induction. Therefore, heuristics to propose such lemmas are essential. A systematic (but necessarily incomplete) approach to introduce new lemmas is to strengthen the conjecture. The strengthened conjecture is proposed as a new lemma, which can then be used to prove the conjecture. Lemmas introduced by strengthening always imply the original conjecture. Thus, it is ensured that a successful proof of them is useful, which in turn justifies spending time to prove them.

I show how to integrate generalization, a standard strengthening technique, into SupInd (Sect. 4.4.1). Standard Generalization picks a complex term in the conjecture and replaces all its occurrences by a fresh variable. In the refutational setting of SupInd generalization picks a complex term in the negated conjecture and replaces its occurrences by a fresh Skolem constant. Generalization operates syntactically and thus it is dependent on the syntactic representation of the conjecture, which, for example, can already be altered by applying rewriting in a different direction. Thus, I also consider if alternative representation (derived from the axioms) of the conjecture are more suitable for generalization.

A shortcoming of generalization is that it ignores dependencies between the generalized term and the remaining conjecture. Term-dependency preserving generalization uses those dependencies to derive side-conditions that lead to better strengthenings (Sect. 4.4.2).

Finally, I describe a new technique, bounded strengthening, to generate helpful propositions, that remove constructor-sink functions (Sect. 4.4.3). Constructor sink-functions are hard for induction, because they often make it impossible to apply the induction hypotheses.

All strengthening techniques are derived from and restrictions of the Derive Facts inference rule, such that unsatisfiablity of the the new (strengthened) state implies unsatisfiablity of the original state. Therefore, I extend SupInd with (specializations of) a strengthening inference rule, which extends the current state instead of requiring a new state with its own search, which the Derive Facts rule requires (to show that the newly derived proposition is a consequence).

Conjecture Strengthening Let S be a simple state and S' be a state such that if S' is inductively unsatisfiable, then S is also inductively unsatisfiable.

$$\frac{S}{S \mid S'} \ \textsc{Strengthen}$$

Conjecture Strengthening Is a Derived Rule The Conjecture Strengthening rule can be derived from the SupInd calculus presented earlier. If there is a proof using the SupInd and the Conjecture Strengthening rules, then there is proof using only the SupInd rules. Suppose the proof contains a refutation of S (i.e. the left-hand side of the $|$-node introduced), then the strengthening step can be simply removed. Suppose the proof does not contains a refutation of S, then it contains a refutation of S'. S' is only inductively unsatisfiable if S is inductively unsatisfiable, because of the definition of the Conjecture Strengthening rule. Thus, S' is an consequence of S. Therefore, by the Derive Facts rule the clauses in S' can be added to S and thus the proof for S' can be used for the extended S.

Completeness For refutational completeness (nested) strengthenings are not sufficient, because without an unrestricted Derive Facts inference not all possibly useful propositions are generated. An enumeration based on an unrestricted Derive Facts inference can be interleaved, i.e. performed only once in a while, with the in practice more useful strengthening techniques to retain completeness.

There are approaches to enumerating all propositions for the equational case. For example HipSpec/QuickSpec [17] uses a depth-limited testing-based generation of equations. The approach works by building equivalence classes of terms using testing and seems to be useful in practice. However, how to extend this approach to work in a clausal setting (e.g. conditional equations), is still an open question.

Nonetheless, I expect that restricting enumeration to strong strengthening techniques is superior for a large number of practical applications, because those strengthening techniques, while generating a wide range of propositions, only generate propositions whose proof is relevant to the proof of the current (sub)conjectures.

4.4.1. Generalization in SupInd

The intuition behind generalization [16, 22, 59] is that by replacing a complex term of the conjecture by a fresh Skolem constant, the corresponding induction hypotheses become more powerful. It is particular useful on conjectures containing the same (complex) term multiple times.

Superposition (and in turn SupInd) uses a powerful reduction mechanism to simplify all clauses. This means in particular that all negated conjecture clauses are simplified and that the non-simplified clauses are removed from the simple state. Unfortunately, for many (negated) conjectures some non-simplified clauses are a better basis for generalization, than the simplified versions present in the simple state. This is an important point often overlooked for generalization and is especially important for SupInd because of the powerful reduction mechanism of superposition (which is necessary for efficient first-order reasoning).

Consider the conjecture $\mathit{rev}(\mathit{rev}(Xs)) \approx Xs$ from the IsaPlanner benchmark [22] and the axioms $N_A =$

$$\left\{\begin{array}{lll} (1) & \mathit{app}(\mathit{nil}, Ys) \approx Ys & \\ (2) & \mathit{app}(\mathit{cons}(X, Ys_1), Ys_2) & \approx \; \mathit{cons}(X, (\mathit{app}(Ys_1, Ys_2))) \\ (3) & \mathit{rev}(\mathit{nil}) \approx \mathit{nil} & \\ (4) & \mathit{rev}(\mathit{cons}(X, Ys)) & \approx \; \mathit{app}(\mathit{rev}(Ys), \mathit{cons}(X, \mathit{nil})) \end{array}\right\}$$

and the negated conjecture:

$$\mathit{rev}(\mathit{rev}(\mathsf{ys})) \not\approx \mathsf{ys}$$

After applying the induction inference once, the base-case $\mathsf{ys} = \mathit{nil}$ is trivially proven and the simplified negated conjecture for the step-case $\mathsf{ys} = \mathit{cons}(\mathsf{x}, \mathsf{xs})$ is

$$N_C = \{\ (5) \quad \mathit{rev}(\mathit{app}(\mathit{rev}(\mathsf{xs}), \mathit{cons}(\mathsf{x}, \mathit{nil}))) \not\approx \mathit{cons}(\mathsf{x}, \mathsf{xs})\ \}$$

together with the induction hypothesis

$$N_{IH} = \{\ (6) \quad \mathit{rev}(\mathit{rev}(\mathsf{xs})) \approx \mathsf{xs}\ \}$$

Unfortunately, we are unable to use the induction hypothesis, because there is no (first-order reasoning) way to eliminate the app between the two occurrences of rev. With a suitable ordering, no further clauses are inferred at all, which means this simple state is not first-order refutable. Applying generalization to the only clause (5) of the negated conjecture does not progress towards a proof. Nonetheless, there are consequences of the clauses (1) to (6) that contain the same complex subterm several times (at different positions). Thus, they would be suitable candidates for generalization. The following clauses are consequences of the clauses (1) to (6):

$$\begin{array}{lll}
\text{(a)} & \mathit{rev}(\mathit{app}(\mathit{rev}(\mathsf{xs}), \mathit{cons}(\mathsf{x}, \mathit{nil}))) & \not\approx \mathit{cons}(\mathsf{x}, \mathsf{xs}) \\
\text{(b)} & \mathit{rev}(\mathit{app}(\mathit{rev}(\mathsf{xs}), \mathit{cons}(\mathsf{x}, \mathit{nil}))) & \not\approx \mathit{cons}(\mathsf{x}, \mathit{app}(\mathit{nil}, \mathsf{xs})) \\
\text{(c)} & \mathit{rev}(\mathit{app}(\mathit{rev}(\mathsf{xs}), \mathit{cons}(\mathsf{x}, \mathit{nil}))) & \not\approx \mathit{cons}(\mathsf{x}, \mathit{app}(\mathit{rev}(\mathit{nil}), \mathsf{xs})) \\
\text{(d)} & \mathit{rev}(\mathit{app}(\mathit{rev}(\mathsf{xs}), \mathit{cons}(\mathsf{x}, \mathit{nil}))) & \not\approx \mathit{app}(\mathit{cons}(\mathsf{x}, \mathit{rev}(\mathit{nil})), \mathsf{xs}) \\
\text{(e)} & \mathit{rev}(\mathit{app}(\mathit{rev}(\mathsf{xs}), \mathit{cons}(\mathsf{x}, \mathit{nil}))) & \not\approx \mathit{app}(\mathit{rev}(\mathit{cons}(\mathsf{x}, \mathit{nil})), \mathsf{xs}) \\
\text{(f)} & \mathit{rev}(\mathit{app}(\mathit{rev}(\mathsf{xs}), \mathit{cons}(\mathsf{x}, \mathit{nil}))) & \not\approx \mathit{app}(\mathit{cons}(\mathsf{x}, \mathit{nil}), \mathsf{xs}) \\
\text{(g)} & \mathit{rev}(\mathit{app}(\mathit{rev}(\mathsf{xs}), \mathit{cons}(\mathsf{x}, \mathit{nil}))) & \not\approx \mathit{cons}(\mathsf{x}, \mathit{rev}(\mathit{rev}(\mathsf{xs})))
\end{array}$$

Clauses (e), (f) and (g) contain a term multiple times and can be generalized. But clauses (a) to (g) are redundant with respect to clauses (1) to (6) and thus are never introduced by first-order reasoning. The following clauses are generalizations of the clauses (e), (f) and (g):

$$\begin{array}{llll}
\text{(e}'\text{)} & \mathit{rev}(\mathit{app}(\mathit{rev}(\mathsf{xs}), \mathsf{xs}_3)) & \not\approx & \mathit{app}(\mathit{rev}(\mathsf{xs}), \mathsf{xs}_3) \\
 & \quad\text{generalizing term} & & \mathit{cons}(\mathsf{x}, \mathit{nil}) \text{ to } \mathsf{xs}_3 \\
\text{(f}'\text{)} & \mathit{rev}(\mathit{app}(\mathit{rev}(\mathsf{xs}), \mathsf{xs}_4)) & \not\approx & \mathit{app}(\mathsf{xs}, \mathsf{xs}_4) \\
 & \quad\text{generalizing term} & & \mathit{cons}(\mathsf{x}, \mathit{nil}) \text{ to } \mathsf{xs}_4 \\
\text{(g}'\text{)} & \mathit{rev}(\mathit{app}(\mathsf{xs}_5, \mathit{cons}(\mathsf{x}, \mathit{nil}))) & \not\approx & \mathit{cons}(\mathsf{x}, \mathit{rev}(\mathsf{xs}_5)) \\
 & \quad\text{generalizing term} & & \mathit{rev}(\mathsf{xs}) \text{ to } \mathsf{xs}_5
\end{array}$$

Clauses (e$'$) and (f$'$) are the negation of non-theorems and are eventually removed (Sect. 4.5). Clause (g$'$) is the negation of a theorem ($\mathit{rev}(\mathit{app}(Xs, \mathit{cons}(\mathsf{x}, \mathit{nil}))) \approx \mathit{cons}(\mathsf{x}, \mathit{rev}(Xs))$) and can be automatically refuted (proven). The generalization (g') is a strengthening of the negated conjecture (clause (5)), therefore refuting the generalization (g') also refutes the negated conjecture (clause (5)). Since clause (5) represents the negated conjecture of the step case of the initial conjecture ($\mathit{rev}(\mathit{rev}(XS)) \approx XS$) and the proof of the base case is trivial, the initial conjecture can be automatically shown.

We have just seen that for generalization to be useful, it must also be applied to clauses that would normally not be present in a simple state's negated conjecture. To be able to find generalizations, we introduce an inference rule (NEWEQ), which applies the axioms, specifying the defined functions, in the inverse direction to that used for superposition and rewriting, generating additional candidates on which generalization can be performed:

$$\frac{C' \vee s' \# s[u]}{C' \vee s' \# s[t']} \text{ NEWEQ} \quad \text{if} \quad t \approx t' \in N_A \cup N_{IH} \cup N_C$$

where $\# \in \{\approx, \not\approx\}$, $u = t\sigma$, $t'\sigma \not\prec t\sigma$ and $\prec$ is the reduction ordering of the superposition calculus.

Since the rule NEWEQ potentially generates an infinite number of clauses, it is applied only a fixed number of times per (sub)term. In Pirate we generate all combinations where it is applied at most once per subterm. With this limit, clauses (a) to (g) are all the clauses that are generated by NEWEQ from the negated conjecture (clause 5). NEWEQ is not added to the calculus, but only used within generalization to generate additional candidates. The process is as follows: First candidates are generated by applying NEWEQ rule to some combination of terms in the conjecture. The Generalization rule then introduces new simple states for each term that occurs multiple times in one of the candidates. The new state's conjectures are created by replacing the term that occurs multiple times by a fresh Skolem constant. The following inference rule is a special case of the Strengthening rule and can therefore be used to extend SupInd by generalization:

Generalization Let N be a clause set constructed by applying NEWEQ to N_C, let t be term that occurs in N. Let sk be a fresh Skolem constant and A be $N_A \cup N_{IH} \cup N_C$.

$$\frac{(N_A,\, N_{IH},\, N_C)}{(N_A,\, N_{IH},\, N_C) \mid (A,\, \emptyset,\, N[t \mapsto sk])} \text{ GEN}$$

In Pirate we apply the Generalization rule exhaustively to the original state each time we apply the Structural Induction inference rule. We apply it to all terms that occur in the negated conjecture (N_C) or at least twice in a NEWEQ-extension (N) of the negated conjecture (N_C) of the simple state that the induction rule was applied on.

4.4.2. Term-Dependency Preserving Generalization

Despite its usefulness, generalization does not consider dependencies between different terms. Let P be an arbitrary predicate, len the length and rev the reverse of a list and the conjecture be $P(len(Xs)) \rightarrow P(len(rev(Xs)))$ and thus let the negated conjecture be

$$P(len(\mathrm{xs})) \wedge \neg P(len(rev(\mathrm{xs})))$$

Generalization could replace $len(\mathrm{xs})$ to x_1 deriving the generalized negated conjecture:

$$P(\mathrm{x}_1) \wedge \neg P(len(rev(\mathrm{xs})))$$

or $rev(\mathrm{xs})$ to x_2, $len(rev(\mathrm{xs}))$ to x_3 or some combination. However, in all combinations the connection between $len(\mathrm{xs})$ and $len(rev(\mathrm{xs}))$ is lost. Losing such a connection typically means deriving a non-theorem. The problem here is how to generate a generalization that does not lose any connections between different occurrences of the same Skolem function.

The solution is to prove that some contexts of the different occurrences of the Skolem function are equal. In the example the different contexts of the only Skolem function xs are:

$$\mathsf{xs} \qquad \mathit{len}(\mathsf{xs}) \qquad \mathit{rev}(\mathsf{xs}) \qquad \mathit{len}(\mathit{rev}(\mathsf{xs}))$$

The only well-typed combinations are xs with $\mathit{rev}(\mathsf{xs})$ and $\mathit{len}(\mathsf{xs})$ with $\mathit{len}(\mathit{rev}(\mathsf{xs}))$, deriving the following two propositions:

$$(1)\ \mathit{rev}(Xs) \approx Xs \qquad (2)\ \mathit{len}(\mathit{rev}(Xs)) \approx \mathit{len}(Xs)$$

Both justify the generalization of the original negated conjecture to

$$P(\mathsf{x}) \land \neg P(\mathsf{x})$$

The resulting generalization is trivial to prove. That it is solvable at all is due to the fact that the dependency between both occurrences of xs is preserved by the freshly introduced generalization Skolem constant x. To ensure that the generalization implies the original conjecture, one of the side-condition propositions must be proven. While proposition (1) is not a theorem, proposition (2) can be shown by induction and first-order reasoning. I show how to detect and remove non-theorems from SupInd's search in Sect. 4.5.

In general, a combination of side-condition propositions can be used together to rewrite multiple terms of the negated conjecture to generate the term-dependency preserving generalization. Several of such side-condition combinations can derive the same term-dependency preserving generalization and thus a proof of the generalization and one of the combinations of side-conditions is sufficient. Parts of the conjecture that are not rewritten in order to derive the generalization, can be kept also for the side conditions. As an example consider the (non-negated) conjecture:

$$Q \lor P(t_1, t_2)$$

This conjecture can be generalized to

$$Q \lor P(X, X)$$

as long as both the following side conditions hold:

$$Q \lor t_1 \approx s \qquad\qquad Q \lor t_2 \approx s$$

From $Q \lor t_1 \approx s$ and $Q \lor t_2 \approx s$ we can derive $Q \lor t_1 \approx t_2$, which together with $Q \lor P(X, X)$ implies $Q \lor P(t_1, t_2)$.

For SupInd we have to consider the negated conjecture, where the same approach is possible. The negated conjecture which we have to refute that corresponds to the conjecture $Q \lor P(t_1, t_2)$ is

$$\neg Q \land \neg P(t_1, t_2)$$

This negated conjecture can be generalized to

$$\neg Q \land \neg P(X, X)$$

as long as both the following (non-negated) side conditions hold:

$$Q \vee t_1 \approx s \qquad\qquad Q \vee t_2 \approx s$$

where the corresponding negated conjecture for the side conditions are

$$\neg Q \wedge t_1 \not\approx s \qquad\qquad \neg Q \wedge t_2 \not\approx s$$

This approach is captured by the following inference rule that infers one term-dependency preserving generalization using different combinations of side-condition propositions. All are combined into one state that tracks their dependencies. Term-Dependency Preserving Generalization is special case of the Strengthening rule.

Term-Dependency Preserving Generalization Let N_l and N_r be clause sets such $N_l \uplus N_r = N_C$. Let N be a set of clauses constructed by applying *NewEq* to N_r. Let $t_1, \ldots, t_n$ be terms that occur in N and contain Skolem functions. Let sk be a fresh Skolem constant and let A be $N_A \cup N_{IH} \cup N_C$.

$$\frac{(N_A,\, N_{IH},\, N_C)}{(N_A,\, N_{IH},\, N_C) \mid ((A,\, \emptyset,\, N_l \cup N[t_1 \mapsto sk,\, \ldots,\, t_n \mapsto sk]) \;\&\; (SideC_1 \mid \cdots \mid SideC_m))}\ \textsc{DepGen}$$

Each set of side conditions $SideC_i$ is defined as

$$(A,\, \emptyset,\, N_l \cup s_{1_l} \not\approx s_{1_r}) \;\&\; \cdots \;\&\; (A,\, \emptyset,\, N_l \cup s_{o_l} \not\approx s_{o_r})$$

such that $t_1, \ldots, t_n$ are equal after rewriting with instances of the equations $s_{1_l} \approx s_{1_r}, \ldots, s_{o_l} \approx s_{o_r}$.

In Pirate we apply the Term-Dependency Preserving Generalization rule whenever we apply Generalization. In Pirate Term-Dependency Preserving Generalization is implemented separately of (but dependent on) Generalization for historical reasons. For all Skolem functions (sk_t) occurring in the conjecture we find potential terms ($t_1, \ldots, t_n$) such that they are the first or second superterms (of the same data type as sk_t) of some occurrence of sk_t.

4.4.3. Strengthening Conjectures with Bounds

Generalization and term-dependency preserving generalization are heuristics and thus alone not sufficient to prove every type of conjecture. In particular, they are less useful in cases where functions occur whose return value is always a subterm of one of its argument values. Examples of such functions are the *minimum* and *subtraction* of natural numbers and the unconditional *drop* for lists (*drop* removes the N first elements from a list). I call those functions constructor-sink functions, because they remove constructors from terms they are applied to. Constructor-sink functions generally lead to conjectures where the induction hypothesis cannot be applied and are thus hard to prove without further (inductive) lemmas. As an example, consider such a constructor-sink function, the unconditional *drop* for lists, the conjecture $\mathit{len}(\mathit{drop}(N, Xs)) \leq \mathit{len}(Xs)$, the corresponding negated conjecture:

$$\mathit{len}(\mathit{drop}(\mathsf{n}, \mathsf{xs})) \not\leq \mathit{len}(\mathsf{xs})$$

and the axioms defining the functions *drop* and *len*, and the predicate $\leq$:

$$\begin{array}{lcllcl}
\mathit{drop}(0, Xs) & \approx & Xs & 0 & \leq & Y \\
\mathit{drop}(N, \mathit{nil}) & \approx & \mathit{nil} & s(X) & \not\leq & 0 \\
\mathit{drop}(s(N), \mathit{cons}(X, Xs)) & \approx & \mathit{drop}(N, Xs) & s(X) \leq s(Y) & \leftrightarrow & X \leq Y \\
\mathit{len}(\mathit{nil}) & \approx & 0 & \mathit{len}(\mathit{cons}(X, Xs)) & \approx & s(\mathit{len}(Xs))
\end{array}$$

When performing induction on n and xs the induction cases where n is *0* or xs is *nil* are trivially proven. Consider the negated conjectures for the induction step cases on both n and xs, i.e. where $\mathsf{n} = s(\mathsf{n}')$, $\mathsf{xs} = \mathit{cons}(\mathsf{x}', \mathsf{xs}')$ and $\mathsf{n}' = s(\mathsf{n}'')$, $\mathsf{xs}' = \mathit{cons}(\mathsf{x}'', \mathsf{xs}'')$ and so on:

$$\begin{gathered}
\mathit{len}(\mathit{drop}(\mathsf{n}, \mathsf{xs})) \not\leq \mathit{len}(\mathsf{xs}) \\
\mathit{len}(\mathit{drop}(\mathsf{n}', \mathsf{xs}')) \not\leq s(\mathit{len}(\mathsf{xs}')) \\
\mathit{len}(\mathit{drop}(\mathsf{n}'', \mathsf{xs}'')) \not\leq s(s(\mathit{len}(\mathsf{xs}''))) \\
\vdots
\end{gathered}$$

and the corresponding induction hypotheses:

$$\begin{gathered}
\mathit{len}(\mathit{drop}(\mathsf{n}', \mathsf{xs}')) \leq \mathit{len}(\mathsf{xs}') \\
\mathit{len}(\mathit{drop}(\mathsf{n}'', \mathsf{xs}'')) \leq \mathit{len}(\mathsf{xs}'') \\
\vdots
\end{gathered}$$

Every (combined) induction step on n and xs leads to another constructor (s) accumulating in a non-generalizable position on the right-hand side of the $\leq$ predicate. The nested inductions are futile because the constructor-sink function prevents the application of the induction hypothesis by removing the constructors from left-hand side of the conjecture.

My solution to this problem is to replace (in the negated conjecture) the term containing the constructor-sink function with a term that is less hostile to induction, while ensuring that the resulting conjecture implies the original one.

To find such a term, consider one distinctive feature that many constructor-sink functions share: In all models, the constructor term representing (i.e. that is equal to the interpretation of) the term with a construct-sink function as top-level symbol is a (non-strict) subterm of at least one of the constructor terms representing the argument terms. In the example, the constructor term representing $\mathit{drop}(\mathsf{n}, \mathsf{xs})$ will always be a subterm of the constructor term representing xs. Below we will show a syntactic criterion to automatically determine such “cap subterms” of constructor-sink functions. The first part of my solution is to replace $\mathit{drop}(\mathsf{n}, \mathsf{xs})$ in the negated conjecture by its cap xs. This results in the new conjecture

$$(1)\ \ \mathit{len}(Xs) \leq \mathit{len}(Xs)$$

and its negation, which while seemingly obvious still requires induction on xs:

$$\mathit{len}(\mathsf{xs}) \not\leq \mathit{len}(\mathsf{xs})$$

Alone, the new conjecture (1) does not imply the original one, because the constructor term representing $\mathit{drop}(\mathsf{n}, \mathsf{xs})$ could be a proper subterm of the constructor term representing xs. I rectify this by additionally showing that if the conjecture holds for some constructor term representing $\mathit{drop}(\mathsf{n}, \mathsf{xs})$, it holds for all its immediate constructor subterms and thus for all its constructor subterms. This is done by showing that replacing $\mathit{drop}(\mathsf{n}, \mathsf{xs})$ in the negated conjecture for each step-case constructor (in the case of lists by $\mathit{cons}(\mathsf{y}, \mathsf{ys})$) implies replacing $\mathit{drop}(\mathsf{n}, \mathsf{xs})$ by each of the step-constructors immediate subterms of the same type (in the case of $\mathit{cons}(\mathsf{y}, \mathsf{ys})$ by ys).

This leads to the new side-condition conjecture

$$(2)\quad \mathit{len}(\mathit{cons}(Y, Ys)) \leq \mathit{len}(\mathit{cons}(X, Xs)) \implies \mathit{len}(Ys) \leq \mathit{len}(\mathit{cons}(X, Xs))$$

and its negated conjecture:

$$\mathit{len}(\mathsf{ys'}) \not\leq \mathit{len}(\mathsf{xs}) \wedge \mathit{len}(\mathit{cons}(\mathsf{y}, \mathsf{ys'})) \leq \mathit{len}(\mathsf{xs})$$

Together the new conjectures (1) and (2) imply the original conjecture. Both conjecture (1) and conjecture (2) are provable by one induction inference and first-order reasoning. The conjectures are easier, mainly because the constructor-sink function was removed.

I now properly define the ideas that were presented in the above example.

Constructor Normal Form A term of the form $f(t_1, \ldots, t_n)$ of type τ is in *constructor normal form* if and only if τ is a data type, f is a constructor of that data type and all t_i that are of type τ are in constructor normal form. By the restricted semantics of data types, it follows that for every term t of a data type τ and each interpretation $\mathcal{I}$ there is exactly one term t' in constructor normal form such that $\mathcal{I}(t) = \mathcal{I}(t')$. I call such a t' the *constructor normal form* of t.

Cap Subterm A *cap subterm* is a proper subterm s of a term t such that the constructor normal form of t is a (non-strict) subterm of the constructor normal form of s. A *cap argument position* of a function symbol f is an argument position i of f such that for every term $f(t_1, \ldots, t_m)$ and every interpretation it is guaranteed that t_i is a cap subterm of $f(t_1, \ldots, t_m)$.

Exhaustive Recursion To determine a syntactic criterion for cap argument positions, it is necessary to find sets of axioms that specify one function symbol, so that all models have to obey that specification. In principle, we want universally quantified axioms about such a function symbol f, i.e. axioms of the form $f(X_1, \ldots, X_n) \approx t$. For data types we know that all elements of a data type are build by constructors (i.e. 'no-junk') thus we can also use all axioms of the form $f(X_1, \ldots, \mathit{nil}, \ldots, X_n)$ and $f(X_1, \ldots, \mathit{cons}(Y, Ys), \ldots, X_n)$ as long as each position is exhaustively specified (e.g. by a complete pattern matching on all constructors of a data type).

I restrict the definitions to top-level constructor symbols for each argument. An extension to arbitrary depth and number of constructor symbol is straight forward, but adds no further insights while significantly complicating the presentation.

Let F be a set of conditional equations $C_i \vee f(t_{1_i}, \ldots, t_{n_i}) \approx t_{r_i}$ where each C_i is a (possibly empty) disjunction of literals and each t_{j_i} is either a universally quantified variable or a term consisting only of a top-level constructor symbol with universally quantified variables as arguments. I further assume that for the purpose of this section, corresponding variables in different conditional equations are actually the same variable, e.g. that N in the defining equations of gcd (its defining equations are given below) always denotes the same variable. I also assume, that the t_{j_i} are distinct. If the t_{j_i} are not distinct, we can make them distinct and add equality constraints to the corresponding condition C_i. Furthermore, for each t_{j_i} that is not a variable let τ_i be its data type and for each constructor f_j of τ_i there must be a at least one t_{j_i} that is equal to $f_j(X_1, \ldots, X_m)$. Let J_{f_j} be the union of all C_i such that t_{j_i} is equal to $f_j(X_1, \ldots, X_m)$ then it must holds that $\neg(C_1 \vee \cdots \vee C_w)$ is unsatisfiable (where each $C_k \in J_{f_j}$). If that is the case then the function symbol f is called *exhaustively recursive*.

The constructor-sink functions drop, minus and $\mathit{subtraction}$ all are exhaustively recursive. So is the greatest common divisor of two numbers gcd, if the conditions $N \approx M$ (from equation 1), $N < M$ (equation 2) and $M < N$ (equation 3) are exhaustive, i.e. if $\neg(N \approx M \vee N < M \vee M < N)$ can be shown to be unsatisfiable:

$$\begin{array}{lll}
\text{(gcd equation 1)} & \mathit{gcd}(N, M) \approx N & \text{if } N \approx M \\
\text{(gcd equation 2)} & \mathit{gcd}(N, M) \approx \mathit{gcd}(M, N) & \text{if } N < M \\
\text{(gcd equation 3)} & \mathit{gcd}(N, M) \approx \mathit{gcd}(N - M, M) & \text{if } M < N
\end{array}$$

A Syntactic Criterion for Cap Argument Positions Let f be a exhaustively-recursive function symbol whose defining set of equations (ignoring potential side-conditions C_i) can be partitioned into two sets, the recursive equations and the non-recursive equations.

1. $f(t_1, \ldots, t_n) \approx t_r$ (non-recursive, i.e. t_r does not have f as top symbol)
2. $f(t_1, \ldots, t_n) \approx f(r_1, \ldots, r_n)$ (recursive)

A term t_c is a *constructor subterm* (*c-subterm*) of a term t if t_c is syntactically equal to t or if $t = f_i(t_1, \ldots, t_n)$, t is of the data type τ, f_i is a constructor of τ, for some i it holds that t_i is of the same type as t (i.e. τ) and t_c is a c-subterm of t_i. For all models and terms t_c and t it holds that if t_c is a c-subterm of t then t_c's constructor normal form is a (non-strict) subterm of t's constructor normal form.

Each of the defining equations of f can be mapped to a set of argument positions where the right-hand side is a c-subterm of an argument of the left-hand side or to a set of positions, where the argument of the right-hand side is a c-subterm of the corresponding argument on the left-hand side: To derive a simple syntactic criterion for cap argument positions, I first map the equations to a set of argument positions. Then the intersection of those sets contains the argument positions that are cap argument positions. The mapping distinguishes between the both forms of equations that we consider:

1. $f(t_1, \ldots, t_n) \approx t_r \mapsto \{i \mid t_r \text{ is a c-subterm of } t_i\}$ (non-recursive)
2. $f(t_1, \ldots, t_n) \approx f(r_1, \ldots, r_n) \mapsto \{i \mid r_i \text{ is a c-subterm of } t_i\}$ (recursive)

The intersection of these sets contains only cap argument positions of f. This is because only positions that remove constructors, from top-most argument term at the position, are in the sets created by the mapping. I.e. only positions where (in the non-recursive case) the result (t_r) is a c-subterm of the argument position or (in the recursive case) the same position, in the recursive call, is a c-subterm.

The computation of cap argument positions can be strengthened by allowing permutations in the recursive definitions, within those argument positions that are cap argument positions. This is achieved by first assuming the intersection of argument positions of the non-recursive equations are cap argument positions. Then, iteratively, argument positions are removed that do not preserve c-subterms under permutation in the recursive equations. The cap argument positions are those that are still in the set when a fixpoint is reached:

1. CAP_0 ::= intersection of all $f(t_1,\dots,t_n) \approx t_r \mapsto$
   $\{i \mid t_r \text{ is a c-subterm of } t_i\}$ (non-recursive)

k. CAP_k ::= intersection of CAP_{k-1} and the intersection of all $f(t_1,\dots,t_n) \approx f(r_1,\dots,r_n) \mapsto$
   $\{i \mid i \in CAP_{k-1} \wedge \exists j.\ r_i \text{ is a c-subterm of } t_j\}$ (recursive)

This can be further strengthened by not requiring c-subterms but also allowing cap subterms. This means replacing the conditions "t_r is a c-subterm of t_i" by "t_r is a c-subterm of t_i or t_r has a cap subterm that is a c-subterm of t_i" and "r_i is a c-subterm of t_j" by "r_i is a c-subterm of t_j or r_i has a cap subterm that is a c-subterm of t_j". Furthermore, for increased prevision, the equality constraints of the conditions C_i of each defining equation can be taken into account when testing if terms are c-subterms of each other.

This syntactic criterion is not complete, i.e. it does not detect all cap argument positions, but it is sufficient for many practically interesting cases. In particular, it is sufficient for all constructor-sink functions mentioned so far. For example for subtraction ($-$):

1. $0 - M \approx 0 \mapsto \{1\}$ because 0 is a c-subterm of 0[1].
   $N - 0 \approx N \mapsto \{1\}$ because N is a c-subterm of N and not a c-subterm of 0.

2. $s(N') - s(M') \approx N' - M' \mapsto \{1\}$ because $2 \notin CAP_1$ and N' is a c-subterm of $s(N')$.

3. Fixpoint reached for subtraction ($-$): $\{1\}$

The fixpoint results is correct, because position 1 is indeed a cap argument position for subtraction ($-$). The cap-argument position computation for the greatest common divisor $gcd(N,\ M)$ is as follows:

1. $gcd(N,\ M) \approx N \mapsto \{1,\ 2\}$ when considering the side condition that $N \approx M$

2. $gcd(N,\ M) \approx gcd(M,\ N) \mapsto \{1,\ 2\}$ because N c-subterm of N and M c-subterm of M
   $gcd(N,\ M) \approx gcd(N-M,\ M) \mapsto \{1,\ 2\}$ because $N - M$ has the cap subterm N which is a c-subterm of N and M is a c-subterm of M

[1]Because 0 is a constant and the only base-case constructor, it can be considered as a c-subterm of M, in that case $0 - M \approx 0 \mapsto \{1,\ 2\}$. CAP_1 would still be $\{1\}$ because of the other non-recursive equation

3. Fixpoint reached for $gcd(N, M)$: $\{1, 2\}$

Indeed, both argument positions are cap-argument positions for the greatest common divisor (*gcd*).

The Bound Strengthening Inference As an initial preprocessing step, all exhaustively recursive functions of the initial axiom set and their cap argument positions are determined by the syntactic criterion introduced above.

The Bound Strengthening inference itself then checks if the conjecture of a simple state contains a term t_1 that has a cap subterm t_2. Then it creates several new simple states. The first simple state is generated by replacing t_1 by its cap subterm t_2. The remaining simple "Inverse Step" states are then generated to guarantee that the new simple state implies the original one. Because t_2 is a cap subterm of t_1, we know that the constructor term corresponding to the interpretation of t_1 is a subterm of the constructor term corresponding to the interpretation of t_2. Therefore, the simple "Inverse Step" states must ensure that removing (top-level) constructors from the position t_2 occurs in the conjecture is possible. This ensures that all constructor terms from the base-cases up to the cap subterm t_2 fulfill the original conjecture.

The following inference rule extends SupInd to support strengthenings by using known upper-bounds and (several) inverse induction steps. It is a special case of the Strengthening rule.

Bound Strengthening Let t_1 be a term of a data type, let t_2 be a cap subterm, let $\kappa_1, \ldots, \kappa_n$ be the data type's constructors and let A be $N_A \cup N_{IH} \cup N_C$.

$$\frac{(N_A,\, N_{IH},\, N_C)}{(N_A,\, N_{IH},\, N_C) \mid ((A,\, \emptyset,\, N_C[t_1 \mapsto t_2])\ \&\ Step_{\kappa_1}\ \&\ \cdots\ \&\ Step_{\kappa_n})}\ \textsc{Bound}$$

Let a be the arity of κ_i. Then $Step_{\kappa_i}$ is defined as

$$Step_{\kappa_i,1}\ \&\ \cdots\ \&\ Step_{\kappa_i,a}$$

where $Step_{\kappa_i,j}$ is defined as $(\{\bot\},\{\bot\},\{\bot\})$ if argument j of κ_i is not of the same type as κ_i. Otherwise, it is the inverse induction step for argument j of constructor κ_i and defined as

$$(A,\, \emptyset,\, cnf(\neg N_C[t_1 \mapsto \kappa_i(\mathsf{sk}_1,\, \ldots,\, \mathsf{sk}_m)]) \cup N_C[t_1 \mapsto \mathsf{sk}_j])$$

where $\mathsf{sk}_1, \ldots, \mathsf{sk}_m$ are fresh Skolem constants and $cnf(\neg N_C[t_1 \mapsto \kappa_i(\mathsf{sk}_1, \ldots, \mathsf{sk}_m)])$ computes the clause normal form of the negation of $N_C[t_1 \mapsto \kappa_i(\mathsf{sk}_1, \ldots, \mathsf{sk}_m)]$.

In Pirate we apply the Bound Strengthening rule exhaustively to the original simple state each time we apply the Structural Induction inference rule. We apply it to all terms of the simple state's N_C that have an cap subterm.

Lemma 4.4.3.1 (The Bound Strengthening Rule is Sound).
If the result of the bound strengthening rule $(N_A, N_{IH}, N_C) \mid ((A, \emptyset, N_C[t_1 \mapsto t_2])\ \&\ Step_{\kappa_1}\ \&\ \cdots\ \&\ Step_{\kappa_n})$ is unsatisfiable, then the original state (N_A, N_{IH}, N_C) is also unsatisfiable.

Proof. A $|$-state is unsatisfiable if one of its substates is unsatisfiable. If the left substate is unsatisfiable, then obviously the original state is also unsatisfiable, because they are identical.

It remains to show that if the right substate of the $|$-state is unsatisfiable, then the original state is unsatisfiable. Suppose $((A,\, \emptyset,\, N_C[t_1 \mapsto t_2])\ \&\ \mathit{Step}_{\kappa_1}\ \&\ \cdots\ \&\ \mathit{Step}_{\kappa_n})$ is unsatisfiable but $(N_A,\, N_{IH},\, N_C)$ is not. The negated conjecture of each $\mathit{Step}_{\kappa_i,j}$, for each constructor κ_i and each j which is an argument position of t_1 of the same data type, is

$$\mathit{cnf}(\neg N_C[t_1 \mapsto \kappa_i(\mathsf{sk}_1,\, \ldots,\, \mathsf{sk}_m)]) \cup N_C[t_1 \mapsto \mathsf{sk}_j]$$

Without changing the satisfiability, we can remove the clause normal transformation and use $\wedge$ to signify $\cup$, because the clause set represents a conjunction of the clauses it contains

$$\neg N_C[t_1 \mapsto \kappa_i(\mathsf{sk}_1,\, \ldots,\, \mathsf{sk}_m)] \wedge N_C[t_1 \mapsto \mathsf{sk}_j]$$

The corresponding non-negated conjecture for each $\mathit{Step}_{\kappa_i,j}$ is

$$N_C[t_1 \mapsto \kappa_i(X_1,\, \ldots,\, X_m)] \implies N_C[t_1 \mapsto X_j]$$

Each of these $\mathit{Step}_{\kappa_i,j}$ follows from $N_A \cup N_{IH} \cup N_C$ and thus must also hold in every interpretation of $(N_A,\, N_{IH},\, N_C)$. Therefore, $((A,\, \emptyset,\, N_C[t_1 \mapsto t_2])$ must be unsatisfiable and $(N_A,\, N_{IH},\, N_C)$ must not be unsatisfiable for $((A,\, \emptyset,\, N_C[t_1 \mapsto t_2])\ \&\ \mathit{Step}_{\kappa_1}\ \&\ \cdots\ \&\ \mathit{Step}_{\kappa_n})$ to be unsatisfiable but $(N_A,\, N_{IH},\, N_C)$ not to be unsatisfiable. The only possible way for all $\mathit{Step}_{\kappa_i,j}$ to hold and $((A,\, \emptyset,\, N_C[t_1 \mapsto t_2])$ to be unsatisfiable is therefore if the constructor normal form of t_2 is not a (non-strict) subterm of the constructor normal form of t_1. Because t_2 is a c-subterm of t_1, this is not possible. $\square$

4.5. An Optimized Variant of SupInd

In this section, I describe two additions to SupInd, that make an implementation more efficient. First, I introduce a global axiom clause set (Sect. 4.5.1) and then show how to detect and remove invalid propositions (Sect. 4.5.2)

4.5.1. A Shared Global Axiom Clause Set

In the SupInd calculus presented so far the initial axiom set is duplicated for each simple state. This is wasteful and unnecessary, because inferences within that clause set are repeatedly performed in each simple state. An alternative is to introduce a global axiom clause set (N_{GA}) that extends every simple state's clause sets so that inferences between the axioms are only performed once. The inferences of the superposition calculus (lifted to SupInd) must then also be applied to N_{GA} and the result of those inferences must be added back to N_{GA}. SupInd's Superposition inference is therefore replaced by the following two inference rules.

The new status of an SupInd proof state is represented by the pair $(N_{GA},\, S)$, where N_{GA} is the global axiom clause set and S is state. The pair representing the proof state is unsatisfiable if the global axiom clause set N_{GA} contains the empty clause or the state S is unsatisfiable. Simple states are still the triples $(N_A,\, N_{IH},\, N_C)$, where N_A now contains results of local first-order reasoning instead of the global axioms and their reasoning. Let N_A be the initial axioms and N_C be the initial negated conjecture then the initial proof state is $(N_A,\, (\emptyset,\, \emptyset,\, N_C))$. The only inference rule that directly alters the global axioms clause set is the superposition rule for that clause set.

Optimized Conjecture Superposition for Global Axioms
Let N_{GA} be a clause set, S be a state and $C_1, C_2 \in N_{GA}$.

$$\frac{(N_{GA},\, S)}{(N_{GA} \cup \{C'\},\, S)}\ \textsc{Inf}_{GA} \quad \text{if} \quad \frac{C_1 \quad C_2}{C'} \ \text{ or } \ \frac{C_1}{C'} \quad \text{is a superposition inference}$$

For a simpler and more uniform presentation, I still present the remaining inference rules as if they operate only on a state and leave the global axiom clause set implicit, if necessary I refer to it as N_{GA}. The Superposition inference rule of SupInd must be replaced by superposition rules that take the global axiom clause set into account. First, the Conjecture Superposition inference rule, that performs the inferences between all clause sets and the clause sets for the the axioms (N_A) and negated conjecture (N_C). The inference results of this inference go to the axiom clause set N_A. The clause sets for axioms and the negated conjecture are still separate to achieve smaller induction hypotheses in the Structural Induction inference.

Optimized Conjecture Superposition
Let $n \in \{1,2\}$, let $C_1 \in N_A \cup N_C$ and let $C_2 \in N_{GA} \cup N_A \cup N_{IH} \cup N_C$.

$$\frac{(N_A,\, N_{IH},\, N_C)}{(N_A \cup \{C'\},\, N_{IH},\, N_C)}\ \textsc{Inf}_{C} \quad \text{if} \quad \frac{C_1 \quad C_2}{C'}\ ,\ \frac{C_2 \quad C_1}{C'} \ \text{ or } \ \frac{C_1}{C'} \quad \text{is a superposition inference}$$

And then the inference rule that does inferences within the induction hypotheses clause set (N_{IH}) with help of the global axiom clause set (N_{GA}), where the inference results can stay in the induction hypotheses clause set.

Optimized Induction Hypotheses Superposition
Let $n \in \{1,2\}$, let $C_1 \in N_{IH}$ and let $C_2 \in N_{GA} \cup N_{IH}$.

$$\frac{(N_A,\, N_{IH},\, N_C)}{(N_A,\, N_{IH} \cup \{C'\},\, N_C)}\ \textsc{Inf}_{IH} \quad \text{if} \quad \frac{C_1 \quad C_2}{C'}\ ,\ \frac{C_2 \quad C_1}{C'} \ \text{ or } \ \frac{C_1}{C'} \quad \text{is a superposition inference}$$

The rules ensure that inferences are only applied with clauses from the simple state's clauses or with a clause from the simple state and a clause from the a global axiom clause set. An additional benefit of splitting the Superposition rule this way is that less inferences within N_{IH} must be repeated, after an induction inference. Inference results from (global) axioms and the induction hypotheses are preserved (and adapted) in N_{IH} by the induction inference.

The Redundancy Elimination rule must also be changed to take N_{GA} into account and is therefore replaced by

Optimized Redundancy Elimination
Let N, N' be sets of clauses such that N is reducible to N' with respect to $N_{GA} \cup N_{IH} \cup N'$.

$$\frac{(N_A \uplus N,\ N_{IH},\ N_C)}{(N_A \cup N',\ N_{IH},\ N_C)}\ \textsc{Red}'_A \qquad \frac{(N_A,\ N_{IH} \uplus N,\ N_C)}{(N_A,\ N_{IH} \cup N',\ N_C)}\ \textsc{Red}'_{IH} \qquad \frac{(N_A,\ N_{IH},\ N_C \uplus N)}{(N_A,\ N_{IH},\ N_C \cup N')}\ \textsc{Red}'_C$$

With these changes, the duplication of inferences between the global axioms can be avoided, because they are only performed once in the global axiom clause set.

4.5.2. Purging Invalid Propositions

Detecting invalid propositions is useful to reduce the search space by removing the corresponding simple states. It also helps end-users to detect erroneous conjectures. For example Pirate can automatically detect that one problem (problem 87) of the IsaPlanner benchmark is not a theorem:

$$\mathit{minus}(\mathit{minus}(X,\ Y),\ Z) \approx \mathit{minus}(X,\ \mathit{minus}(Y,\ Z))$$

Here, *minus* is subtraction over the natural numbers. None of the tools described in the evaluation section Sect. 4.7 is capable of detecting that this is not a theorem. To the best of my knowledge, including a non-theorem in the benchmark was not intentional.

In the presence of induction, a first-order saturation of the axioms and the negated conjecture is not sufficient to show that the conjecture is invalid. There could be additional lemmas (provable by induction) that, when added to the previously saturated set, yield a contradiction. A test that is still useful in the presence of induction is: A conjecture is invalid if its negation follows from a consistent set of axioms. In the case of superposition this can be restated as follows: a conjecture is invalid if its negation reduces to the consistent axioms. Furthermore, we know that if the axioms are consistent and a proposition is valid with respect to the axioms, then the union of every of the proposition's induction hypotheses with the axioms is consistent.

Theorem 4.5.2.1 (Invalid Conjecture)**.**
If the negated conjecture clause set N_C of a simple state is empty then the corresponding conjecture is invalid.

If the global axiom clause set is not consistent, the Optimized Conjecture Superposition for Global Axioms inference rule of SupInd will eventually derive the empty clause, thereby proving the initial conjecture. Therefore, the individual inference rules of the optimized variant of SupInd dealing with states can always assume that the global axiom clause set is consistent.

If the negated conjecture set (N_C) of a simple state is the empty set, it is implied by the axioms, induction hypothesis and inference results. Thus, the conjecture cannot hold. Because consistency of the axioms can be assumed and because induction hypotheses of valid propositions together with a consistent set of axioms cannot be unsatisfiable, simple states with an empty conjecture (N_C) can be removed. Thus, optimized SupInd can be extended by the following simplification rules.

Optimized Purge Invalid Propositions Let S be an arbitrary state.

$$\frac{(N_A,\, N_{IH},\, \{\})\ \&\ S}{(\{\},\, \{\},\, \{\})}\ \&_l \qquad \frac{S\ \&\ (N_A,\, N_{IH},\, \{\})}{(\{\},\, \{\},\, \{\})}\ \&_r \qquad \frac{(N_A,\, N_{IH},\, \{\})\mid S}{S}\ |_l \qquad \frac{S \mid (N_A,\, N_{IH},\, \{\})}{S}\ |_r$$

4.6. Implementation

In this section I give an overview over the way the SupInd calculus is implemented in the version of Pirate that supports induction. There are three main data structures that hold all necessary information:

N_{GA} the global axiom clause set. Initially, this set holds all input axioms. During the proof search all those inductive consequences that Pirate has proved to follow directly from the global axiom clause set are added.

tree the and/or-tree representing the state. The simple states, which are all at the leaves and contain the clause set triples, are not part of the and/or-tree but only referenced by it. Simplifying the tree means to propagate the (dis)proofs. An or-node of the tree is proven if one of its subnodes is proven and disproven if all its subnodes are disproven. An and-node of the tree is proven if all of its subnodes are proven and disproven if one of its subnodes are disproven. In keeping with SupInd states I also refer to the or-nodes as |-states and the and-nodes as &-states.

states a list of simple states annotated with additional information that is useful in a practical implementation. One such information is if further induction is necessary or only first-order reasoning is required and the other main information is a reference counter for each state, counting how often that state is referenced in the and/or-tree. This list of simple states is deduplicated so that a simple-state can occur multiple time in the *tree* but only occurs once in the *states*.

Initially, the global axiom clause set is filled with the input axioms. The negated conjecture N_C is represented by new simple state $(\emptyset,\, \emptyset,\, N_C)$ that is added to the *states* list. The *tree* is initialized with a single reference to the currently only simple state. Optionally, Pirate has a setting so that it starts with attempts to proof associativity and commutativity for all binary function symbols, before starting the actual proof attempt.

4.6.1. The Main Loop

After initialization the main loop (Alg. 1) is started. At the beginning of each loop iteration the next simple-state is selected (line 2). Then superposition is performed on the global axiom

Algorithm 1: *SupInd Main Loop*

```
1  while states is not empty and tree is not a proof or a disproof do
2      nextState := extract the next state from states;
3      Sup($N_{GA}$);   /* Sect. 4.6.2 */
4      Sup(nextState);   /* Sect. 4.6.2 */
5      simplify the tree accordingly if a (dis)proof is found;
6      move globally valid conjectures to $N_{GA}$;   /* I.e. for proofs without side conditions */
7      add nextState to end of states;
8      if nextState is not marked as first-order reasoning only then
9          replace nextState in tree by an |-state with the following children:
               • itself, flagged to only continue first-order reasoning;
               • Induction(nextState);   /* Sect. 4.6.3 */
               • Generalization(nextState);   /* Sect. 4.6.4 */
               • Bound Strengthening(nextState);   /* Sect. 4.6.6 */
10 return tree /* tree now contains the proof derivation or the disproof */;
```

clause set and the selected simple-state (lines 3-4). If this results in a proof or disproof the *tree* is updated, and the corresponding states are removed from *states* (line 5). If a proof was obtained and the conjecture is not dependent on any side conditions (e.g. the N_A and N_{IH} parts of the simple state are empty), then the conjecture can be added to the global axiom clause set N_{GA} (line 6). The simple state is added to the end of the *states* list (line 7). If the simple state requires inductive reasoning, the *tree* and as appropriate the *states* are updated and structural induction and strengthenings are performed (line 9). If the loop terminates the *tree* contains the result and the annotations necessary to reconstruct a proof.

4.6.2. Superposition

Superposition is applied to the global axiom set and the union of the clause set triple of the currently processed simple state. In addition to the reduction rules implemented for the typed calculus (Sect. 3.4.1), a restricted from of splitting is implemented. Superposition is run with a time limit and a limit on the number of inferences that are performed. The global axiom set and the simple states are updated, i.e. the resulting inferences are added to them. If an empty clause is derived, i.e. the current state is shown unsatisfiable, then the *tree* is updated accordingly. Additionally, the first-order reasoning steps are recorded for the later proof output. Similarly, when the current conjecture is shown to be invalid, the state is removed from the tree. If the parent-state is an &-state, that &-state is also removed. No recording of reasoning steps is necessary in this case.

Distinctness and Injectivity

For the superposition calculus used in SupInd we integrated special reduction rules to for distinctness (Def. 4.2.1.4) and injectivity (Def. 4.2.1.5). The main motivation for this is to ensure that distinctness and injectivity properties are used to reduce a clause as soon as possible. Additionally, the integration leads to more compact and readable proofs, because the application of those properties is immediately obvious instead of hidden behind other reduction rules.

Distinctness of data type constructors means that clauses that contain a disequation between to distinct constructors are true. Furthermore, equations between two distinct constructors can be removed from clauses. The rules treating injectivity for negative and positive literals of clauses are as follows.

Distinct Let f_i and f_j be data type constructors such that $i \neq j$.

$$\frac{f_i(\ldots) \not\approx f_j(\ldots) \vee C}{\top}\ \textsc{Neg Distinct} \qquad \frac{f_i(\ldots) \approx f_j(\ldots) \vee C}{C}\ \textsc{Pos Distinct}$$

Injectivity of data type constructors means that disequation between to distinct constructors are true only if one of the arguments is not equal. Furthermore, equations between two distinct constructors can only be true if all arguments are equal. The rules treating injectivity for negative and positive literals of clauses are as follows.

Injective Let f_i be a data type constructor.

$$\frac{f_i(s_1, \ldots, s_n) \not\approx f_i(t_1, \ldots, t_n) \vee C}{s_1 \not\approx t_1 \vee \ldots \vee s_n \not\approx t_n \vee C}\ \textsc{Neg Inj.} \quad \frac{f_i(s_1, \ldots, s_n) \approx f_i(t_1, \ldots, t_n) \vee C}{(s_1 \approx t_1 \vee C)\ \wedge\ \ldots\ \wedge\ (s_n \approx t_n \vee C)}\ \textsc{Pos Inj.}$$

I also added two special rules that handles the (dis)equation of variables (only for single literal clauses). Two variables of a data type with more than one constructor cannot be equal for all instantiations. Such single literal clauses are often introduced by using injectivity, therefore special reduction rules seem advisable.

Data Type Variable Let X, Y be variables of a data type, with more than one constructor.

$$\frac{X \not\approx Y}{\top}\ \textsc{Neg Vars} \qquad \frac{X \approx Y}{\bot}\ \textsc{Pos Vars}$$

4.6.3. Structural Induction

To perform the Structural Induction inference rule the current goal is analyzed first, in order to determine the Skolem constant c whose induction cases we want to generate next (line 1, Sect. 4.3). Then the goals for the induction cases including the induction hypotheses are generated (lines 3-9, Sect. 4.2). The induction hypotheses will not be created for base cases since there is no argument of the same data type (lines 5-7). Previous induction hypotheses are kept for all cases (line 8). Note that in order to create the proper induction hypotheses, only the clauses from the negated conjecture N_C, i.e. those clauses that represent the (potentially reduced) negated conjecture are 'unnegated' (line 6). Because of previous reasoning steps, the negated conjecture of the simple state can be different from the initial negated conjecture of the simple state. Thus, the additional negation step is required.

Algorithm 2: Structural Induction

```
   Input: $N_A$ axioms, $N_H$ hypotheses, $N_C$ conjectures,
1  $\mathsf{c}$    := analyze($N_A$,$N_H$,$N_C$);  /* Sect. 4.3 */
2  cases := new &-state;
3  foreach constructor $f$ of the data type of $\mathsf{c}$ do
4      let $\mathsf{c}_1,\ldots,\mathsf{c}_n$ be fresh Skolem constants such that $f(\mathsf{c}_1,\ldots,\mathsf{c}_n)$ is well-typed;
5      foreach $\mathsf{c}_i$ of the same data type as $\mathsf{c}$ do
6          $N'_H$ := unnegate($N_C[\mathsf{c} \mapsto \mathsf{c}_i]$) and clausify;
7          $N'_H$ += $N_H[\mathsf{c} \mapsto \mathsf{c}_i]$;
8      $N'_H$ += $N_H[\mathsf{c} \mapsto f(\mathsf{c}_1,\ldots,\mathsf{c}_n)]$;
9      cases += ($N'_H$,$N_C[\mathsf{c} \mapsto f(\mathsf{c}_1,\ldots,\mathsf{c}_n)]$);
10 return cases;
```

unnegate negates the negated conjecture to retrieve the current conjecture from the negated conjecture. When unnegating, all Skolem constants, except the Skolem constant c of the current induction step, can be treated as existentially quantified and thus are turned into universally quantified variables. Then the result is clausified again to initialize the new induction hypotheses (N'_H, line 6). In Pirate I have also implemented an alternative unnegate which preserves the Skolem constants, yielding less powerful ground induction hypotheses which have easier first-order goals.

Induction hypotheses of strong structural induction hold for all structurally smaller terms. In each induction case ($f(\mathsf{c}_1,\ldots,\mathsf{c}_n) \approx \mathsf{c}$), the arguments ($\mathsf{c}_i$) of the data type constructor (f) are structurally smaller than the whole constructor term. Thus all induction hypotheses that hold for c also hold for c_i and the corresponding induction hypotheses are added (line 7). This way the (generally infinite) set of terms smaller than c and their induction hypotheses are gradually constructed in subsequent induction steps (then over the c_i) when the Skolem constants representing those smaller terms are introduced.

4.6.4. Generalization

Generalization first preprocesses the simple state by generating a set of negated conjectures, which are different representations of the negated conjecture (N_C) of the simple state generated by applying the NEWEQ inference rule (line 2, Sect. 4.4.1). The generated set contains the negated conjecture (N_C) and all negated conjectures that can be derived by applying NEWEQ at most once per subterm of the negated conjecture. Then, for each generated clause we apply generalization to every complex term that occurs at least once in N_C or more than once in a NEWEQ-extension (line 3). The complex terms are then replaced by fresh Skolem constants in the negated conjecture, the axioms and the induction Hypotheses(lines 5-7). Furthermore, new simple states representing the generalizations are created (line 8). Finally, the dependency preserving generalization is performed using the information gathered by the standard generalization (line 9).

```
Algorithm 3: Generalization
   Input: $N_A$ axioms, $N_{IH}$ hypotheses and $N_C$ negated conjectures clauses of a simple-state
1  generalizations := new |-state;
2  foreach $n_C$ in Preprocess($N_A$, $N_H$, $N_C$) do
3      commonterms := compute set of subterms of $n_C$ that occur at least twice in $n_C$ (at
       least once if $n_C$ is $N_C$);
4      foreach $S \subseteq$ commonterms do
5          $n'_C := c[t_1 \mapsto b_t, \ldots, t_n \mapsto b_t$ for each $t_i \in S]$ for some fresh Skolem constants $b_t$;
6          $n'_A := N_A[t_1 \mapsto b_t, \ldots, t_n \mapsto b_t$ for each $t_i \in S]$
7          $n'_{IH} := N_{IH}[t_1 \mapsto b_t, \ldots, t_n \mapsto b_t$ for each $t_i \in S]$
8          generalizations += $(n'_A, n'_{IH}, n'_C)$;
9          generalizations += DepPreserve($N_A$, $N_{IH}$, $N_C$, $n'_C$, $b_t$);   /* Sect. 4.4.2 */
10 return generalizations;
```

4.6.5. Term-Dependency Preserving Generalization

Term-Dependency Preserving Generalization (the inference rule is described in Sect. 4.4.2) requires as input a simple state, a term (t) contained in the negated conjecture and a fresh Skolem constant (b_t). The simple state is the one which contains the negated conjecture we are using to generalize. The term t is the term that generalization picked to generalize and the Skolem constant b_t is by what the term t was replaced. Then, for each Skolem constant occurring in the term t term-dependency preserving generalization is performed (line 2). First, the negated conjecture is split into an unchanging part (N_l) and a generalization part (N_r) according to the Term-Dependency Preserving Generalization inference rule (line 3). In our evaluations (Sect. 4.7) the unchanged part N_l, in the generalizations used in successful proofs, is empty. This is because most negated conjectures of the evaluation only have one or two clauses. From the generalization part (N_r) of the negated conjecture a set of terms is selected that contain the Skolem constant b_t (line 4). In Pirate we consider all first or second superterms (of the same data type as b_t) of some occurrence of b_t. Then an &-state that contains both the term-dependency preserving generalization and the side conditions is created (lines 5-9). The generalization is created by replacing all chosen superterms (S) by the generalization Skolem constant b_t (line 6). For each such superterm ($t_i \in S$) the side condition that it must be equal to t is required. This is ensured by adding the appropriate simple state (representing the conjecture that $t_i \approx t$) to the &-state (line 8).

Algorithm 4: *DepPreserve*

```
   Input: $N_A$ axioms, $N_{IH}$ hypotheses and $N_C$ negated conjectures clauses of a simple-state,
          $t$ term generalized in goal, $b_t$ Skolem constant to which $t$ was generalized;
1  newgoals := new |-state;
2  foreach c in Skolem-symbols-of(t) do
3      $(N_l, N_r)$ := split $N_C$;
4      foreach $S$ in choose-set-of-superterms-of-c in $N_r$ do
5          DependencyLemma := create &-state;
6          DependencyLemma += $(N_A, \emptyset, N_l \cup N_r[t_1 \mapsto b_t, \ldots, t_n \mapsto b_t$ for each $t_i \in S])$;
7          foreach $t_i$ in $S$ do
8              DependencyLemma += $(N_A, \emptyset, N_l \cup \{t_i \not\approx t\})$;
9          newgoals += DependencyLemma;
10 return newgoals;
```

4.6.6. Strengthening Conjectures with Bounds

Strengthening conjectures with bounds has two phases and its inference rule is described in section 4.4.3. The first is an initial phase to gather cap argument positions of all function symbols. The second phase is run each time a negated conjecture is strengthened.

Initial Analysis Phase

The first phase determines which function symbols have cap argument positions. This is achieved by initially (before starting the SupInd main loop) computing the structural analysis fixpoint for each function symbol. The fixpoint starts with the assumption that all positions of all functions

Algorithm 5: *Cap Argument Position Analysis*

```
   Input: $N_A$ axioms
1  positions := $[f \mapsto \{1, \ldots, arity(f)\} \mid$ for all function symbols $f]$;
2  foreach c in $N_A$ do
3      foreach $f(t_1, \ldots, t_n) \approx t$ in c do
4          foreach i in $\{1, \ldots, n\}$ do
5              if $t == f(r_1, \ldots, r_n)$ and $r_i$ is not c-subterm of $t_i$ and
                       $r_i$ has no cap subterm that is a c-subterm of some $t_i$
                       such that $i \in$ positions$(f)$ then
6                  positions$(f)$ := positions$(f) \setminus \{ i \}$;
7              else if $t_r$ is not c-subterm of some $t_i$ and
                       $t_r$ has no cap subterm that is a c-subterm of some $t_i$ then
8                  positions$(f)$ := positions$(f) \setminus \{ i \}$;
9  return positions;
```

are structurally bigger than their result (line 1). The algorithm then proceeds by iterating over all defining equations (which are marked in the input) to check if they obviously uphold the subterm property (line 3-8). To that end, the right hand side must either be a recursive call to the same function, such that all arguments of the recursive call are smaller than some argument of the initial call (such that that argument is still marked as a cap argument position, lines 5-6). Alternatively, the complete right hand side (t_r) can be smaller than some argument t_i (lines 7-8). This algorithm assumes that the marked initial axioms fully define the used functions.

Strengthening Conjectures

The second phase generalizes a negated conjecture according to the information gathered by the analysis phase. For each negated conjecture that is to be strengthened, each (sub)term (t) that has a cap subterm is taken as a basis for strengthening (line 2). For each cap subterm t_c of t the Bound Strengthening inference is applied (line 3). The inference results in an &-state consisting of the strengthened conjecture and the inverse steps (lines 4-9). The strengthening is derived by replacing t by t_c in the negated conjecture (N_C, line 5). For each constructor of the data type of t an inverse step is necessary (line 6). Each inverse step ensures that if the conjecture holds, with t replaced by each constructor with fresh Skolem constants as argument, it also holds for t replaced with the Skolem constant arguments of the same data type (lines 7-9).

Algorithm 6: *Strengthening Conjectures with Bounds*

```
   Input: $N_A$ axioms, $N_{IH}$ hypotheses and $N_C$ negated conjectures clauses of a simple-state
1  strengthenings := new |-state;
2  foreach $t$ occurring in $N_C$ and $t$ is of a data type do
3      foreach $t_c$ that is a cap subterm of $t$ do
4          boundstates := new &-state;
5          boundstates += $(N_A, \emptyset, N_C[t \mapsto t_c])$;
6          foreach $\kappa_i$ constructor of $t$'s data type do
7              $t_\kappa := \kappa_i(s_1, \ldots, s_n)$ for fresh Skolem constants $s_i$;
8              foreach $s_i$ that is of $t$'s data type do
9                  boundstates += $(N_A, \emptyset, cnf(\neg N_C[t_i \mapsto t_\kappa]) \cup N_C[t_i \mapsto s_i])$;
10         strengthenings += boundstates;
11 return strengthenings;
```

4.6.7. Example

In this section I show one example file and Pirate's output on that file. The file is in the DFG format. It is the 38th problem of the IsaPlanner [22] benchmark set (Sect. 4.7).

Input

The input of Pirate is the DFG file format used for the monomorphic SPASS [9], extended with polymorphism and data type declarations. In the preamble, the DFG format has a number of descriptions, that we do not use in Pirate. They are still present in the input file, because the DFG format requires them.

```
begin_problem(benchmark).

list_of_descriptions.
name({**}).
```

```
author({**}).
status(unknown).
description({**}).
end_of_list.
```

The function, predicates and type symbols are declared after the preamble. For function and predicate symbols, the format is **(***name-of-the-symbol***,** *number-of-type-arguments***+***number-of-term-arguments***)**. If the number of type arguments is zero, the number and the **+** can be omitted. For types only the number of type arguments is needed, for background compatibility types can also be named sorts.

```
list_of_symbols.
functions [(nil, 1+0), (cons, 1+2), (z,0), (s,1), (append, 1+2), (count,1+2)].
predicates [(less, 1+2)].
sorts [(list,1),(nat,0)].
end_of_list.
```

Next come the function and predicate declarations ($\mathcal{F}$, $\mathcal{P}$). The format of the declarations begins with **function(** or **predicate(**. Then comes the function-symbol (which must have been defined above), followed by a comma. The next component is the type variable declarations **[**A,...**]**, list(A)), again followed by a comma. The number of type variables must agree with the declared number of type arguments. The last component is a tuple of the arguments followed by white space and the return type (only for functions). The number of argument types must agree with the declared number of term arguments. The declaration is finished by a closing bracket and a dot.

The data type declarations ($\mathcal{DT}$) are also declared here. The format of the data type declarations begins with **datatype(** and is followed by the type constructor that build the data type (e.g. nat) and a common. Next is a comma separated list of constructors. The declaration is finished by a closing bracket and a dot.

```
list_of_declarations.
function(nil,  [A], list(A)).
function(cons, [A], (A,list(A)) list(A)).
function(z, nat).
function(s, (nat) nat).
function(append, [A], (list(A),list(A)) list(A)).
function(count,  [A], (A,list(A)) nat).

predicate(less,  [A], A, A).

datatype(nat, z, s).
datatype(list, nil, cons).
end_of_list.
```

After the declarations, the next part of the input are the axiom formulas. The formulas have a name attached to them, so that the proof output can refer to the formulas by name. The logical part of the formulas is build from type quantification (**forall_sorts**), term quantification (**forall**) and the usual connectives (**equal**, **implies**,...). Terms are build from variables or the function symbol with first the type arguments $< \cdots >$ followed by the term arguments in round brackets. The **:lr** annotation (for equations and equivalences), informs the solver that the equations should be oriented from left to right. We introduced it in previous work on SPASS [9].

```
list_of_formulae(axioms).
formula(forall_sorts([A], forall([LS:list(A)],
          equal:lr(append<A>(nil<A>,LS),      LS))),                  append_nil).

formula(forall_sorts([A], forall([LS:list(A), Y:A, YS:list(A)],
          equal:lr(append<A>(cons<A>(Y,YS),LS),
                   cons<A>(Y,append<A>(YS,LS))))),                   append_cons).

formula(forall_sorts([A], forall([X:A, LS:list(A)],
          equal(count<A>(X,nil<A>),      z))),                        count_nil).

formula(forall_sorts([A], forall([X:A, L:A, LS:list(A)],
          implies(equal(X,      L),
                   equal(count<A>(X,cons<A>(L,LS)),
                         s(count<A>(X,LS)))))),                      count_cons_equal).

formula(forall_sorts([A],forall([X:A, L:A, LS:list(A)],
          implies(not(equal(X,      L)),
                   equal(count<A>(X,cons<A>(L,LS)),
                         count<A>(X,LS))))),                         count_cons_notequal).
end_of_list.
```

The last part of the input is the conjecture. Superposition will negate the conjecture before trying to show the axioms and the conjecture to be unsatisfiable.

```
list_of_formulae(conjectures).
formula(forall_sorts([A],forall([X:A, XS:list(A)],
          equal(count<A>(X,append<A>(XS,cons<A>(X,nil<A>))),
                s(count<A>(X,XS)))))),                               conj_prop38).
end_of_list.

end_problem.
```

Output

I now show the output of the inductive version of Pirate on the problem file presented above. First the preamble is printed. The *reduceDFGtoFacts* script is used to replay first-order proof obligations in the monomorphic version of SPASS. This is done during development to ensure that the proofs found are actually correct. I did not use the replay during evaluation.

```
Starting
scala version 2.11.2
Unkown/Legacy-style Argument: /home/dwand/pirate/reduceDFGtoFacts
```

The negated, clausified and Skolemized conjecture is printed next. Clauses always start with a hash of the clause (here: FB832), followed by a counter that is independent of the clause structure (00007). Next comes the priority (rank) of the clause. Axioms have priority 1000 while the negated conjecture has priority 1. Newly inferred clauses have a priority one higher than the minimum priority of their parents. Then, the tab-separated list of negative literals is printed. The output **->** separates the list of positive literals from the negative literals. Finally, the origin is printed. It refers to input formulas by their name and to previous clauses by their hash.

```
Initial Conjectures:
  : FB832-00007 (     1) {} ||   count(skf1, append(skf2, cons(skf1, nil))) = s(count(skf1, skf2)) ->   .
                                                                  origin(Conj(conj_prop38 from: Set(B94AA)))
```

After printing the conjecture, the proof search is started. This proof is relatively simple and therefore one few outputs were made. Every time a new globally valid axiom (i.e. without any assumptions) is found it is printed. Optionally, its proof can also be printed. For this proof, Pirate discovered that *nil* is also a neutral argument to *append* if its in the second argument position. The CaseProofTask represent the states of the SupInd calculus. At the beginning of the line, after the @, the time in milliseconds is shown (since program start).

```
@        233  @head(1 rank: 10 ): CaseProofTask-TID:4( parent TID:3, DeriveCase( ->   append(V2, nil) = V2 . ))
Welcoming new knowledge:
  : 5A0D0-00105 ( 1000) {} || -> append(V2, nil) lr=> V2* .                              origin(Derived(E60F3))
@        326  @head(0 rank: 12 ): CaseProofTask-TID:6( parent TID:5, Initial)
```

After the successful proof search , either a proof or a counter example is found. Otherwise, a time out occurs. If a proof or counter example was found, it is printed. For a proof, the axiom and conjecture formulas that were used are also listed.

```
Found false
Formulas used:  conj_prop38 count_cons_equal append_nil count_nil append_cons count_cons_notequal
```

Finally, the remaining output consists of the full refutation proof (without clausification). The clauses are printed in the order they were derived, which for smaller proofs (like this one) gives a good separation into the induction cases. The conclusion of the refutation proof is the last clause derived and thus the last clause printed.

The first part of the proof output are the clausified input clauses and the initial reduction results.

```
Involved clauses:
B94AA-00001 ( 1000) {} || ->  count(V2, append(V4, cons(V2, nil))) = s(count(V2, V4)) .  origin(Input(conj_prop38))
6E897-00002 ( 1000) {} || ->  append(nil, V2) = V2 .                                       origin(Input(append_nil))
761AF-00003 ( 1000) {} || ->  append(cons(V2, V4), V6) = cons(V2, append(V4, V6)) .      origin(Input(append_cons))
CE930-00004 ( 1000) {} || ->  count(V2, nil) = z .                                          origin(Input(count_nil))
E3A4E-00005 ( 1000) {} || V2 = V4 ->  count(V2, cons(V4, V6)) = s(count(V2, V6)) .  origin(Input(count_cons_equal))
0CF07-00006 ( 1000) {} || ->  count(V2, cons(V4, V6)) = count(V2, V6) V2 = V4 .  origin(Input(count_cons_notequal))
FB832-00007 (    1) {} || count(skf1, append(skf2, cons(skf1, nil))) = s(count(skf1, skf2)) ->  .
                                                                 origin(Conj(conj_prop38 from: Set(B94AA)))
6F4F1-00010 ( 1000) {} || V2 = V2 ->  count(V2, cons(V2, V4)) = s(count(V2, V4)) .              origin(NuV(E3A4E))
96D58-00011 ( 1000) {} || ->  count(V2, cons(V2, V4)) = s(count(V2, V4)) .                    origin(LitElim(6F4F1))
```

After the input clauses, the inference results appear. In this proof they are separated into the induction case for *nil* and *cons* of skf2. The following clause is the negation of the negated conjecture (keeping the Skolemization). It is needed to generate the induction Hypothesis.

```
BC7A5-00153 (    1) {} || ->  count(skf1, append(skf2, cons(skf1, nil))) = s(count(skf1, skf2)) .
                                                                               origin(Negate(Set(FB832)))
```

First, the clauses for the *nil*-case are printed. Because it is a base case, no induction hypothesis is included, thus the clause BC7A5 is only relevant in the step case. The induction case (clause 88E01) is generated by replacing skf2 with *nil* in the initial conjecture (clause FB832)

```
88E01-00156 (    1) {} || count(skf1, append(nil, cons(skf1, nil))) = s(count(skf1, nil)) ->  .
                                                                           origin(IndCase(nil, skf2, FB832))
8E55B-00157 (    1) {} || s(count(skf1, nil)) = s(z) ->  .         origin(Rew(88E01 with Set[CE930, 6E897, 96D58]))
AAE6D-00161 (    1) {} || count(skf1, nil) = z ->  .                                    origin(DT-Injective(8E55B))
BB8E7-00162 (    1) {} || ->  .                                                          origin(MRR(AAE6D,- CE930))
```

Then, the clauses for the *cons*-case are printed. For this case, the induction case (clause 47F39) is generated by replacing skf2 with *cons*(i1003, i1004) in the initial conjecture (clause FB832), where i1003, i1004 are fresh Skolem constants. The induction hypothesis (clause 0A9F1) is generated from the negation of the negated-conjecture (clause BC7A5) by replacing the induction Skolem skf2 by the fresh Skolem i1004, which was created as an argument (of the same type as skf2) to *cons* for the induction case (clause 47F39).

```
47F39-00163 (    1) {} ||  count(skf1, append(cons(i1003, i1004), cons(skf1, nil)))
                           = s(count(skf1, cons(i1003, i1004))) ->  .            origin(IndCase(cons, skf2, FB832))
0A9F1-00165 (    1) {} ||  -> count(skf1, append(i1004, cons(skf1, nil))) = s(count(skf1, i1004)) .
                                                                                  origin(IndHyp(skf2, cons: BC7A5))
4E4C2-00166 (    1) {} ||  count(skf1, cons(i1003, append(i1004, cons(skf1, nil))))
                           = s(count(skf1, cons(i1003, i1004))) ->  .            origin(Rew(47F39 with Set[761AF]))
7D761-00174 (    2) {} ||  count(skf1, append(i1004, cons(skf1, nil))) = s(count(skf1, cons(i1003, i1004)))
                           -> skf1 = i1003 .                                          origin(Sup(4E4C2 by 0CF07))
3E9A2-00177 (    2) {} ||  s(count(skf1, i1004)) = s(count(skf1, cons(i1003, i1004))) -> skf1 = i1003 .
                                                                                  origin(Rew(7D761 with Set[0A9F1]))
3EA25-00186 (    2) {} ||  count(skf1, i1004) = count(skf1, cons(i1003, i1004)) -> skf1 = i1003 .
                                                                                      origin(DT-Injective(3E9A2))
2F309-00242 (    3) {} ||  count(skf1, i1004) = count(skf1, i1004) -> skf1 = i1003    skf1 = i1003 .
                                                                                      origin(Sup(3EA25 by 0CF07))
37EF4-00245 (    3) {} ||  -> skf1 = i1003 skf1 = i1003 .                                   origin(LitElim(2F309))
37EF4-00246 (    3) {} ||  -> skf1 = i1003 .                                                 origin(RemDup(37EF4))
E7E00-00255 (    1) {} ||  s(count(skf1, append(i1004, cons(skf1, nil)))) = s(s(count(skf1, i1004))) ->  .
                                                                           origin(Rew(4E4C2 with Set[37EF4, 96D58]))
D6027-00261 (    1) {} ||  count(skf1, append(i1004, cons(skf1, nil))) = s(count(skf1, i1004)) ->  .
                                                                                      origin(DT-Injective(E7E00))
603D8-00262 (    1) {} ||  ->  .                                                       origin(MRR(D6027,- 0A9F1))
```

Finally, the induction cases are combined and a contradiction is derived between the negated conjecture of the inital property and the clauses derived by induction.

```
F8DD1-00263 (    1) {} ||  ->  .                              origin(Induction(List[cons by 603D8, nil by BB8E7]))
B94AA-00264 ( 1000) {} ||  -> count(V2, append(V4, cons(V2, nil))) = s(count(V2, V4)) .    origin(Derived(F8DD1))
72304-00265 (    1) {} ||  ->  .                                 origin(Contradicting Set(FB832) and Set(B94AA)))
Involved clauses: 29

       Total time: 2.517414314 s
```

More examples can be found in the appendix (Ch. A), including a disproof of the conjecture by finding a counter example (Sect. A.1.4).

4.7. Evaluation and Comparison with Related Work

There are several related inductive theorem proving methods. Some inductive theorem provers' proof methods are based on rewriting and support only equational formulas, e.g. IsaPlanner [22] based on rippling and case-splitting. Some approaches rely on satisfiability modulo theories (SMT): HipSpec [17] and Dafny [42] integrate SMT as a blackbox, whereas the SMT solver CVC4 [50] integrates support for structural induction. SMT is often successful for theorem proving in first-order logic, but typically it does not provide any completeness guarantees. Its interpreted symbols and theories can be beneficial, e.g. dedicated support for natural numbers, can (for larger values) be better than the unary term representation. Other tools provide their own calculus. One example is Zeno [59], whose calculus is based on critical pairs and path, but does not support existential quantification and negation. The interactive proof assistant ACL2 [16] also has powerful induction heuristics. Unlike SMT, superposition-based theorem provers for first-order logic are typically refutationally complete. Thus, it is guaranteed to terminate on an inconsistent first-order clause set, but might diverge on a consistent one. It is used in explicit and consistency based induction methods. Consistency based methods [18], also called inductionless induction, reduce inductive reasoning to first-order consistency checking. While their reduction phase is powerful, finite saturation of first-order theories can often not be obtained. Explicit induction methods, such as structural induction, compute an induction schema at the start of the proof attempt of each conjecture, whereas implicit induction is usually based on term rewrite systems and the induction schema is constructed during the proof attempt [24]. Kersani and Peltier [35] show how an infinite set of (explicit) induction schemata for natural numbers can be integrated into superposition by a loop detection rule. This rule provides syntactic criteria to detect inference loops and derive inductive conclusions from those loops. Our induction method is also based on explicit induction, namely strong structural induction, but our approach also supports other data types and serves as a basis for our other main contributions, i.e. explicit (heuristic) control of where to apply induction and strengthenings for complex conjectures.

A proof of an inductive conjecture might require additional lemmas, which typically are not directly obvious from the axioms and the conjecture. Automatically deriving useful lemmas is crucial to any serious automation of induction [18]. Lemma generation is in principle a variation of the naive idea of enumerating all possible propositions (e.g. by enumerating all formulas). Conjecture strengthening considers only those lemmas that are immediately useful to prove an unproven conjecture. Generalization is an instance of strengthening that is widely used and applicable to arbitrary first-order conjectures [16, 22, 59]. We contributed two other strengthening methods which we believe can be incorporated by most other approaches. Another avenue is to consider all properties that can be guessed from the (initial) problem. One such tool is HipSpec which integrates QuickSpec to compute equalities from terms constructed by declared functions by integrating testing [17]. Those equalities that remain are then proposed as possible lemmas and processed by the inductive prover. CVC4 also includes such a lemma generation module. It is also restricted to (unconditional) equational lemmas.

Since additional lemmas must be proposed and many of the guesses might not be theorems, it is important to purge non-theorems. For explicit superposition based methods it is sufficient to show that the negated conjecture is reduced to the axioms to detect a non-theorem. Saturation itself is not sufficient, because it only shows that the theorem is not a first-order consequence of the axioms. CVC4 uses (syntactic) rule-based filtering of proposed lemmas augmented by instantiation-based testing. Despite the clear benefits of removing non-theorems, surprisingly little research has been published on this.

To evaluate our approach, we use two induction benchmarks. The evaluation results are available under

`http://people.mpi-inf.mpg.de/~dwand/thesis/`

The first benchmark was originally used to evaluate IsaPlanner [22]. In its original form it consists of 87 problems, one of which is not a theorem. We process the problems individually, meaning that we are not reusing results from previous proof attempts. For each problem, we only include the relevant function definitions as axioms. We report the results published for this benchmark for IsaPlanner [22], Zeno [59], HipSpec [17], Dafny [42], ACL2s [16] (as evaluated by Zeno in [59]) and CVC4 [50]. We ran our tool, Pirate, with a single setting on a Xeon E5-2643 3.5 GHz within a Java 8 VM with 80 gigabytes of heapspace and used a timeout of 300 seconds. Pirate completed 83 problems within 10 seconds and 4 problems took up to 200 seconds. Our tool is the only one that can solve the complete benchmark.

Zeno	CVC4	HipSpec	ACL2s	IsaPlanner	Dafny	Pirate
82	80	80	74	47	45	**87**

Even though Zeno reports that the benchmark has only 85 provable problems [59] and HipSpec also only uses those [17], we can prove 86 problems and find a counterexample for the remaining one, which we also count as success. The counterexample is for the 87th problem:

$\mathit{minus}(\mathit{minus}(X,Y),Z) \approx \mathit{minus}(X,\mathit{minus}(Y,Z))$ and is detected by repeated induction steps until X and Z are instantiated to $s(0)$ and Y to 0. The lost (86th) problem is

$$X < Y \rightarrow (\mathit{element}(X,\mathit{insert}(Y,Xs)) \leftrightarrow \mathit{element}(X,Xs))$$

which is provable since X is different from Y if X is strictly smaller than Y. CVC4 is the only other tool capable of proving problem 86 (see Sect. A.1.3 for the full problem and proof output).

To the best of my knowledge, Pirate is currently the only tool capable of disproving problem 87 and proving the complex problem 85 (see Sect. A.1.2 and A.1.4 for the full problem and proof output):

$$\mathit{len}(Xs) \approx \mathit{len}(Ys) \implies \mathit{zip}(\mathit{rev}(Xs),\mathit{rev}(Ys)) \approx \mathit{rev}(\mathit{zip}(Xs,Ys))$$

The other tools either lack support for conditional theorems (HipSpec), do not propose conditional lemmas (IsaPlanner, CVC4), do not support negation (Zeno) or cannot perform nested inductions (Dafny). In our approach there is no fundamental difference between a single literal conjecture or a complex one with multiple clauses.

Many problems from the IsaPlanner benchmark are relatively easy: HipSpec can prove 67 theorems by induction without generating lemmas, whereas we can solve 73 problems the same way. This includes problem 50 (see Sect. A.1.1 for the full problem and proof output):

$$\mathit{butlast}(Xs) \approx \mathit{take}(\mathit{minus}(\mathit{len}(Xs),s(0)),Xs)$$

Neither HipSpec, ACL2s, IsaPlanner nor Dafny can prove it.

The second benchmark set was used by CLAM [33] to evaluate its proof-critics. For a large fraction, 20 of 50 theorems, we find proofs without generating any additional lemmas. A portfolio version of Pirate combining several settings is the most successful on this benchmark. The single most successful setting for Pirate is the one where all features are enabled. There are four main settings for Pirate:

1. Transformation of Skolems into variables in the Structural Induction rule (Sect. 4.2.2)
2. The use of the Skolem selection heuristic (Sect. 4.3)
3. Use of the (term-dependency preserving) generalization (Sect. 4.4.1 and 4.4.2) and/or bound strenghtening (Sect. 4.4.3)
4. Initial guessing of properties (Sect. 4.6)

The portfolio uses three settings:

- the one where all features are enabled (42 problems),
- one where Skolems are not turned into variables in the induction's CNF translation and initially no properties are guessed (+3 problems) and
- one where term-dependency generalization is disabled (+2 problems)

For this benchmark we used a timeout of 2 hours. Pirate completed 34 problems within 1 minute, further 7 within 400 seconds and one problem took 50 minutes. We again report the published numbers in the literature [17, 33, 50].

CLAM	HipSpec	HipSpec portfolio	CVC4	Zeno	Pirate	Pirate portfolio
41	44	**47**	40	21	42	**47**

IsaPlanner and Zeno can certify their proofs with Isabelle. HipSpec and Dafny do not appear to have a detailed proof output. Pirate generates proofs including all first-order reasoning and induction steps. Isabelle's Sledgehammer [10] has support for replaying our first-order steps in Isabelle. We are also interested in integrating into other tools as well, e.g., program verifiers such as Why3 [14].

Currently, there is a focus on creating and extending a new benchmark set called "tons of inductive problems" (TIP) [54]. TIP was officially started at the second workshop for automated induction theorem-proving (WAIT 2015). It currently mainly contains the IsaPlanner and CLAM benchmark and some new benchmark problems. We plan on supporting the TIP format, which is based on the SMT-LIB format [7]. There are some open problems to solve, e.g. the SMT-theories have to be integrated into our superposition based solver. Our current focus is a stronger integration into Isabelle. In particular, support for arbitrary induction schemata and automatic proof reconstruction by Sledgehammer.

5. Conclusion

In the (mechanized) theorem proving community there are two main fields, the automated theorem provers and the proof assistants. Automated theorem provers for first-order logic, like those based on the superposition calculus, provide full, push button, automation. Proof assistants, e.g. for higher-order logic (HOL), like Isabelle/HOL, lack this automation. But they provide abstractions, e.g. type systems, that make them convenient to use for humans. It is generally known in the higher-order research community that no proof assistant can be complete with respect to the standard semantics of HOL. That completeness is only possible for the Henkin semantics [29]. The Henkin semantics restricts the function types in such a way that the logic becomes compact and thus refutational completeness is possible. Furthermore, it seems that tools (and humans) do not construct or find proofs that are possible in the standard semantics, but are not available with the Henkin semantics. Higher order logic, interpreted with the Henkin semantics, can be encoded into first-order logic, in a complete fashion. This motivates an overall research program that attempts to build an automated theorem prover for higher-order logic, by directly supporting the abstractions of proof assistants in an automated theorem prover, instead of encoding them. This thesis presented one step of this research program by making induction and a polymorphic type system with type classes available for superposition in an efficient manner. It thereby enables automated theorem provers that share more features with proof assistants, further closing the gap between the two fields.

I presented a superposition calculus, including all necessary machinery (such as unification and orderings), that incorporates a polymorphic type system extended with type classes. I have shown the polymorphic superposition calculus to be refutationally complete. The only and minor restriction over the untyped calculus is that the typed calculus requires a simplification ordering instead of a reduction ordering that is total on ground terms. This change is caused by the fact that there can now be typed terms that do not share any context. In practice, this is negligible, because the widely used orderings for superposition are simplification orderings, for example both KBO and LPO, are simplification orderings. The polymorphic type system extended with type classes removes all type system related encoding overhead for the HOL family of proof assistants. I have also shown that the presented type system is competitive with monomorphization, especially for larger fact sizes. Monomorphization is generally considered to be an efficient native type encoding. The evaluation shows that the polymorphic type system extended with type classes requires, in average, less time and fewer inferences than the monomorphic and the other type encodings.

I introduced an extension of the typed first-order logic with data types and the SupInd calculus which combines strong structural induction and superposition. An implementation and an evaluation on a set of benchmarks has promising results. To make the SupInd calculus efficient, I developed a heuristic which guides the application of the induction inference into promising directions. This heuristic is complemented by a novel technique that mitigates irrelevant (nested) induction steps, based on the analysis of (partial) proofs. I have shown heuristics on how to generate propositions to discover useful auxiliary lemmas. In contrast to other tools, the heuristics also work in the full (clausal) first-order setting. The heuristics include a novel preprocessing step to the well established generalization. Furthermore, I introduced term-dependency based generalization which discovers stronger generalizations and useful propositions from the required side conditions. My final heuristic to generate auxiliary lemmas is based on the use of an upper bound and is targeted to conjectures that use orderings. I have also presented an optimized variant of SupInd that reduces the number of inference steps that are necessary. The integration between superposition and inductive reasoning also enables the discovery and removal of invalid propositions, which allows us to disprove conjectures. This combination of redundancy and nested induction, without guessing lemmas, is already capable of proving conjectures many other tools struggle with. This is also shown by the promising benchmark results.

5.1. Future Work Directions

This thesis provided one step on bridging the gap between proof assistants and automated theorem provers by providing a type system and support for induction over data type. However, there are other areas of research that remain to be explored until this gap is fully closed.

One such area is the encoding of features of higher-order logic. While implementing the type system has removed one of the encoding phases, the encoding of higher-order features into first-order still remains. This includes encoding λ as combinators, in order to simulate higher-order unification. A superposition calculus that directly supports λ would be interesting. Support for curried functions, e.g. native support for partial applications of functions, would also help efficiency. High-order logic does not differentiate between terms and formulas. While the distinction between those are crucial for superposition, better encodings for boolean connectives and formulas could also help.

Furthermore, the type system could be extended. I have shown refutational completeness for superposition with a polymorphic type system extended by type classes. This type system is sufficient for the HOL family of proof assistants. Supporting dependent type theory would help improve automation for proof assistants such as Coq and Lean and possibly for programming languages such as Scala. A reasonable first step could be support for path-dependent types, like those features in the DOT calculus [1], the underlying and formal description of (a simplified) Scala. Path-dependent types are types that can express (object-oriented) type hierarchies. They strictly subsume polymorphism, but I expect that an approach similar to the one presented here would work for them.

Moving the relevance filtering from the proof assistant to the automated theorem prover by providing all theorems and all open conjectures to the automated theorem prover, could enable the automated theorem prover to do potentially a better job. It would also enable it to preprocess the axioms. This could enable a cooperation between proof assistant and automated theorem prover where automated theorem provers spend significant amounts of time processing the axioms without the user being required to wait on the automated prover. In the current setting the user has to wait for the automated prover to process both the axioms and the conjecture. Preprocessing the axioms can be designed in such a way that it does not require the reasoning over the axioms to start from scratch for every conjecture. Such a integration feature preprocessing would gives an automated theorem prover more time, then the current 30 seconds default in Sledgehammer, to process the axioms. Thus, more of the time the user is waiting can be spent processing the actual conjecture (instead of unchanging axioms).

For structural inductions, detecting and removing invalid properties is important. For users of proof assistants, the information that they are proving a non-theorem can avoid wasted effort. Extending the rather simple detection to finding counter examples used in SupInd, to a more general approach that is also useful for purely first-order settings, would be a nice addition to any integration between proof assistants and automatic tools.

The integration of theories, such as (linear) arithmetic into superposition that also supports a type system would further help automating some tasks that arise in proof assistants. It would also be required to fully support the SMT-based format for the TIP benchmark set. An example of such integration is the hierarchical superposition calculus.

Besides structural induction over data types, another very common induction schema is rule induction, used to proof conjectures over inductive definitions, such as inductively defined sets or predicates. Rule induction is useful, for example, to proof that a property holds for all even numbers, when even numbers are inductively defined. Extending the SupInd calculus further to enable induction over such definitions would further help the integration between proof assistants and automated theorem provers.

Bibliography

[1] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. The essence of dependent object types. In S. Lindley, C. McBride, P. Trinder, and D. Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 249–272. Springer, Cham, 2016.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs: First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, pages 135–150, Berlin, Heidelberg, 2011. Springer.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[4] L. Bachmair and H. Ganzinger. On restrictions of ordered paramodulation with simplification. In M. E. Stickel, editor, *10th International Conference on Automated Deduction: Kaiserslautern, FRG, July 24–27, 1990 Proceedings*, pages 427–441, Berlin, Heidelberg, 1990. Springer.

[5] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[6] L. Bachmair and H. Ganzinger. Equational reasoning in saturation-based theorem proving. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume I, chapter 11, pages 353–397. Kluwer, Dordrecht, The Netherlands, 1998.

[7] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[8] C. Benzmüller, L. C. Paulson, F. Theiss, and A. Fietzke. LEO-II - a cooperative automatic theorem prover for classical higher-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning: 4th International Joint Conference, IJCAR 2008 Sydney, Australia, August 12-15, 2008 Proceedings*, pages 162–170. Springer, Berlin, Heidelberg, 2008.

[9] J. Blanchette, A. Popescu, D. Wand, and C. Weidenbach. More SPASS with Isabelle. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 345–360. Springer, 2012.

[10] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.

[11] J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. *Logical Methods in Computer Science*, 12(4), 2016.

[12] J. C. Blanchette and A. Paskevich. TFF1: The TPTP typed first-order form with rank-1 polymorphism. In M. P. Bonacina, editor, *Automated Deduction – CADE-24: 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 414–420. Springer, Berlin, Heidelberg, 2013.

[13] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing Polymorphism in SMT solvers. In C. Barrett and L. de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo*, volume 367 of *ACM International Conference Proceedings Series*, pages 1–5, 2008.

[14] F. Bobot, J. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Workshop on Intermediate Verification Languages*, 2011.

[15] F. Bobot and A. Paskevich. Expressing polymorphic types in a many-sorted language. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems: 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, pages 87–102. Springer, Berlin, Heidelberg, 2011.

[16] H. Chamarthi, P. Dillinger, P. Manolios, and D. Vroon. The ACL2 sedan theorem proving system. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 291–295. Springer, 2011.

[17] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction (CADE)*, volume 7898 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2013.

[18] H. Comon. Inductionless induction. In Robinson and Voronkov [53], pages 913–962.

[19] Ł. Czajka and C. Kaliszyk. Goal translation for a hammer for Coq (extended abstract). In J. C. Blanchette and C. Kaliszyk, editors, First International Workshop on *Hammers for Type Theories,* Coimbra, Portugal, July 1, 2016, volume 210 of *Electronic Proceedings in Theoretical Computer Science*, pages 13–20. Open Publishing Association, 2016.

[20] L. de Moura. Lost in translation: how easy (automated reasoning) problems become hard due to bad encodings. In *VAMPIRE 2015: The 2nd VAMPIRE Workshop*, 2015.

[21] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340. Springer, Berlin, Heidelberg, 2008.

[22] L. Dixon and J. Fleuriot. Isaplanner: A prototype proof planner in Isabelle. In F. Baader, editor, *CADE*, volume 2741 of *Lecture Notes in Computer Science*, pages 279–283. Springer, 2003.

[23] L. Erkök and J. Matthews. Using Yices as an automated solver in Isabelle/HOL. In *Automated Formal Methods (AFM08)*, pages 3–13, 2008.

[24] S. Falke and D. Kapur. Rewriting induction + linear arithmetic = decision procedure. In *International Joint Conference on Automated Reasoning 2012, Proceedings*, pages 241–255, 2012.

[25] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY, USA, 1993.

[26] F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs: International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, pages 160–174. Springer, Berlin, Heidelberg, 2007.

[27] T. Hailperin. A theory of restricted quantification I. *The Journal of Symbolic Logic*, 22:19–35, 1957.

[28] J. Harrison. HOL Light: An overview. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66, Munich, Germany, 2009. Springer.

[29] L. Henkin. Completeness in the theory of types. *The Journal of Symbolic Logic*, 15(2):81–91, 1950.

[30] J. Herbrand. *Recherches sur la théorie de la démonstration*. 1930.

[31] C. Hritcu, A. Aguirre, C. Keller, and K. Swamy. From F* to SMT. In *1st International Workshop on Hammers for Type Theories*, 2016.

[32] J. Hsiang and M. Rusinowitch. Proving refutational completeness of theorem-proving strategies: The transfinite semantic tree method. *J. ACM*, 38(3):558–586, July 1991.

[33] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.

[34] C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, 2014.

[35] A. Kersani and N. Peltier. Combining superposition and induction: A practical realization. In *FroCoS 2013, Proceedings*, pages 7–22, 2013.

[36] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[37] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. H. Siekmann and G. Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 342–376. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[38] M. Kosta and C. Weidenbach. Automated reasoning, 2012. `http://resources.mpi-inf.mpg.de/departments/rg1/teaching/autrea-ss12/script/script.pdf`.

[39] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In N. Sharygina and H. Veith, editors, *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 1–35. Springer, Berlin, Heidelberg, 2013.

[40] D. Lankford. Canonical inference. In *Technical Report ATP-32*. Dept. of Mathematics and Computer Science, University of Texas, Austin, 1975.

[41] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer.

[42] K. R. M. Leino. Automating induction with an SMT solver. In V. Kuncak and A. Rybalchenko, editors, *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 315–331. Springer, 2012.

[43] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 312–327. Springer, Berlin, Heidelberg, 2010.

[44] W. McCune. OTTER 3.3 reference manual. *CoRR*, cs.SC/0310056, 2003.

[45] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[46] R. Nieuwenhuis and A. Rubio. Theorem proving with ordering and equality constrained clauses. *Journal of Symbolic Computation*, 19(4):321–351, 1995.

[47] T. Nipkow. Order-sorted polymorphism in Isabelle. In *Papers Presented at the Second Annual Workshop on Logical Environments*, pages 164–188. Cambridge University Press, 1993.

[48] T. Nipkow and C. Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 409–418. ACM Press, 1993.

[49] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer, 2002.

[50] A. Reynolds and V. Kuncak. Induction for SMT solvers. In *VMCAI 2015, January 12-14, 2015. Proceedings*, pages 80–98, 2015.

[51] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 133–150. American Elsevier, New York, 1969.

[52] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, Jan. 1965.

[53] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[54] D. Rosén, M. Johansson, and N. Smallbone. Tons of inductive problems, 2015. Available at `https://github.com/tip-org`.

[55] A. Schmidt. Über deduktive Theorien mit mehreren Sorten von Grunddingen. *Mathematische Annalen*, 115(1):485–506, 1938.

[56] M. Schmidt-Schauß. *Computational aspects of an order-sorted logic with term declarations*. PhD thesis, Universität Kaiserslautern, 1988.

[57] S. Schulz. Simple and efficient clause subsumption with feature vector indexing. In M. Bonacina and M. Stickel, editors, *Automated Reasoning and Mathematics*, volume 7788 of *Lecture Notes in Computer Science*, pages 45–67. Springer, 2013.

[58] S. Schulz, L. de Moura, and B. Konev, editors. *4th Workshop on Practical Aspects of Automated Reasoning, PAAR@IJCAR 2014, Vienna, Austria, 2014*, volume 31 of *EPiC Series in Computing*. EasyChair, 2015.

[59] W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In C. Flanagan and B. König, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7214 of *Lecture Notes in Computer Science*, pages 407–421. Springer, 2012.

[60] M. Stickel. The path-indexing method for indexing term. In *Technical Note 473*. Artificial Intelligence Center, SRI International, 1989.

[61] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In A. Bundy, editor, *Automated Deduction — CADE-12: 12th International Conference on Automated Deduction Nancy, France, June 26 – July 1, 1994 Proceedings*, pages 341–355. Springer, Berlin, Heidelberg, 1994.

[62] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[63] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[64] U. Waldmann. Semantics of order-sorted specifications. *Theor. Comput. Sci.*, 94(1):1–35, 1992.

[65] U. Waldmann. Cancellative abelian monoids and related structures in refutational theorem proving (part i). *Journal of Symbolic Computation*, 33(6), June 2002.

[66] U. Waldmann. Cancellative abelian monoids and related structures in refutational theorem proving (part ii). *Journal of Symbolic Computation*, 33(6), June 2002.

[67] C. Weidenbach. Combining superposition, sorts and splitting. In Robinson and Voronkov [53], pages 1965–2013.

[68] M. Wisniewski, A. Steen, and C. Benzmüller. LeoPARD — A generic platform for the implementation of higher-order reasoners. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Intelligent Computer Mathematics: International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, pages 325–330. Springer, Cham, 2015.

A. Example Files with Pirate's Proofs

A.1. IsaPlanner

In this section I present the complete problems mentioned in the chapter on Induction. A commented example (problem 38) can be found in section 4.6.7.

A.1.1. Problem 50

Input

The problem in DFG format, edited for better readability.

```
begin_problem(benchmark).

list_of_descriptions.
name({**}).
author({**}).
status(unknown).
description({**}).
end_of_list.


list_of_symbols.
functions [(z,0), (s,1), (cons, 1+2), (nil, 1+0),
           (minus,2), (len,1+1), (append, 1+2), (take,1+2), (butlast,1+1)].
predicates [(less, 1+2)].
sorts [(list,1), (nat,0)].
end_of_list.

list_of_declarations.
function(z, nat).
function(s, (nat) nat).

function(nil, [A], list(A)).
function(cons, [A], (A, list(A)) list(A)).

function(minus, (nat, nat) nat).
function(len, [A], (list(A)) nat).
function(take, [A], (nat, list(A)) list(A)).
function(butlast, [A], (list(A)) list(A)).

datatype(nat, z, s).
datatype(list, nil, cons).
end_of_list.

list_of_formulae(axioms).
formula(forall([Y:nat, X:nat], equal:lr(minus(z,Y),          z)),          minus_z_left).
formula(forall([Y:nat, X:nat], equal:lr(minus(X,z),          X)),         minus_z_right).
formula(forall([Y:nat, X:nat], equal:lr(minus(s(X),s(Y)),    minus(X,Y))),      minus_s).

formula(forall_sorts([A],      equal:lr(len<A>(nil<A>),      z)),               len_nil).
formula(forall_sorts([A], forall([L:A, LS:list(A)],
         equal:lr(len<A>(cons<A>(L,LS))          s(len<A>(LS))))),      len_cons).

formula(forall_sorts([A], forall([LS:list(A)],
         equal:lr(append<A>(nil<A>,LS),          LS))),                 append_nil).
formula(forall_sorts([A], forall([LS:list(A),Y:A,YS:list(A)],
         equal:lr(append<A>(cons<A>(Y,YS),LS),
                  cons<A>(Y,append<A>(YS,LS)))))),                      append_cons).

formula(forall_sorts([A], equal:lr(butlast<A>(nil<A>),        nil<A>)),      butlast_nil).
formula(forall_sorts([A], forall([X:A],
         equal:lr(butlast<A>(cons<A>(X,nil<A>)),          nil<A>))),   butlast_cons_nil).
formula(forall_sorts([A], forall([X:A, Y:A, LS:list(A)],
         equal:lr(butlast<A>(cons<A>(X,cons<A>(Y,LS))),
                  cons<A>(X,butlast<A>(cons<A>(Y,LS)))))),            butlast_cons_cons).

formula(forall_sorts([A], forall([XS:list(A)],
         equal:lr(take<A>(z,XS),            nil<A>))),                  fact_take_defz).
formula(forall_sorts([A], forall([X:nat, XS:list(A)],
         equal:lr(take<A>(X,nil<A>),        nil<A>))),                  fact_take_defnil).
formula(forall_sorts([A], forall([X:nat, Y:A, YS:list(A)],
         equal:lr(take<A>(s(X),cons<A>(Y,YS)),
                  cons<A>(Y,take<A>(X,YS)))))),                         fact_take_defcons).
end_of_list.

list_of_formulae(conjectures).
formula(forall_sorts([A], forall([XS:list(A), YS:list(A)],
         equal(butlast<A>(XS),
               take<A>(minus(len<A>(XS),s(z)),XS))))),                  conj_prob50).
end_of_list.

end_problem.
```

Output

The boxed parts are the unedited output of Pirate (except for adding bold for keywords). Pirate was using only induction, i.e. no strengthenings.

```
Starting
scala version 2.11.2
Unkown/Legacy-style Argument: /home/dwand/pirate/reduceDFGtoFacts
Initial Conjectures:
  : FE538-00015 (    1) {} ||   butlast(skf1) = take(minus(len(skf1), s(z)), skf1) ->  .              origin(Conj(
        conj_prob50 from: Set(8944B)))
@       252  @head(0 rank: 10 ): CaseProofTask-TID:5( parent TID:4, Initial)
@       452  @head(0 rank: 22 ): CaseProofTask-TID:6( parent TID:5, Case(cons of skf1))

Found false
Formulas used:  len_cons fact_take_defcons minus_z_left conj_prob50 len_nil fact_take_defz fact_take_defnil minus_s
      butlast_nil butlast_cons_nil minus_z_right butlast_cons_cons


Involved clauses:
8944B-00001 ( 1000) {} || ->   butlast(V2) = take(minus(len(V2), s(z)), V2) .          origin(Input(conj_prob50))
36218-00002 ( 1000) {} || ->   minus(z, V2) = z .               origin(Input(minus_z_left))
C3313-00003 ( 1000) {} || ->   minus(V2, z) = V2 .              origin(Input(minus_z_right))
29DEB-00004 ( 1000) {} || ->   minus(s(V2), s(V4)) = minus(V2, V4) .          origin(Input(minus_s))
B7230-00005 ( 1000) {} || ->   len(nil) = z .            origin(Input(len_nil))
E1EC9-00006 ( 1000) {} || ->   len(cons(V2, V4)) = s(len(V4)) .               origin(Input(len_cons))
D0DA3-00009 ( 1000) {} || ->   butlast(nil) = nil .             origin(Input(butlast_nil))
87351-00010 ( 1000) {} || ->   butlast(cons(V2, nil)) = nil .          origin(Input(butlast_cons_nil))
0FD30-00011 ( 1000) {} || ->   butlast(cons(V2, cons(V4, V6))) = cons(V2, butlast(cons(V4, V6))) .
      origin(Input(butlast_cons_cons))
0F9DA-00012 ( 1000) {} || ->   take(z, V2) = nil .              origin(Input(fact_take_defz))
8DC7A-00013 ( 1000) {} || ->   take(V2, nil) = nil .            origin(Input(fact_take_defnil))
30122-00014 ( 1000) {} || ->   take(s(V2), cons(V4, V6)) = cons(V4, take(V2, V6)) .          origin(Input(
      fact_take_defcons))
```

```
FE538-00015 (    1) {} ||       butlast(skf1) = take(minus(len(skf1), s(z)), skf1) ->  .             origin(Conj(
      conj_prob50 from: Set(8944B)))
C9E8C-00110 (    1) {} || ->    butlast(skf1) = take(minus(len(skf1), s(z)), skf1) .           origin(Negate(Set(
      FE538)))
C740D-00112 (    1) {} ||       butlast(nil) = take(minus(len(nil), s(z)), nil) ->  .          origin(IndCase(nil,
      skf1, FE538))
4D0C1-00113 (    1) {} ||       nil = nil ->  .                      origin(Rew(C740D with Set[B7230, D0DA3, 8DC7A,
      36218]))
A2213-00114 (    1) {} || ->  .                    origin(LitElimF(4D0C1))
5CC8F-00115 (    1) {} ||       butlast(cons(i1001, i1002)) = take(minus(len(cons(i1001, i1002)), s(z)), cons(i1001,
       i1002)) ->  .             origin(IndCase(cons, skf1, FE538))
22370-00117 (    1) {} || ->    butlast(i1002) = take(minus(len(i1002), s(z)), i1002) .                  origin(
      IndHyp(skf1, cons: C9E8C))
0E0B3-00118 (    1) {} ||       butlast(cons(i1001, i1002)) = take(minus(len(i1002), z), cons(i1001, i1002)) ->  .
              origin(Rew(5CC8F with Set[E1EC9, 29DEB]))
61994-00119 (    1) {} ||       butlast(cons(i1001, i1002)) = take(len(i1002), cons(i1001, i1002)) ->  .
           origin(Rew(0E0B3 with Set[C3313]))
414EA-00224 (    1) {} ||       butlast(cons(i1001, nil)) = take(len(nil), cons(i1001, nil)) ->  .
      origin(IndCase(nil, i1002, 61994))
4D0C1-00228 (    1) {} ||       nil = nil ->  .                      origin(Rew(414EA with Set[0F9DA, B7230, 87351]))
B6DCB-00229 (    1) {} || ->  .                    origin(LitElimF(4D0C1))
AC725-00230 (    1) {} ||       butlast(cons(i1001, cons(i1003, i1004))) = take(len(cons(i1003, i1004)), cons(i1001,
       cons(i1003, i1004))) ->  .          origin(IndCase(cons, i1002, 61994))
98449-00231 (    1) {} || ->    butlast(cons(i1003, i1004)) = take(minus(len(cons(i1003, i1004)), s(z)), cons(i1003,
       i1004)) .          origin(IndHyp(i1002, cons: 22370))
0F69D-00240 (    1) {} ||       cons(i1001, butlast(cons(i1003, i1004))) = cons(i1001, take(len(i1004), cons(i1003,
      i1004))) ->  .             origin(Rew(AC725 with Set[30122, 0FD30, E1EC9]))
DD524-00241 (    1) {} || ->    butlast(cons(i1003, i1004)) = take(minus(len(i1004), z), cons(i1003, i1004)) .
           origin(Rew(98449 with Set[E1EC9, 29DEB]))
92F72-00242 (    1) {} ||       butlast(cons(i1003, i1004)) = take(len(i1004), cons(i1003, i1004))      i1001 =
      i1001 ->  .          origin(DT-Injective(0F69D))
6B629-00243 (    1) {} ||       butlast(cons(i1003, i1004)) = take(len(i1004), cons(i1003, i1004)) ->  .
           origin(LitElim(92F72))
E98A5-00244 (    1) {} || ->    butlast(cons(i1003, i1004)) = take(len(i1004), cons(i1003, i1004)) .
      origin(Rew(DD524 with Set[C3313]))
F647B-00245 (    1) {} || ->  .                    origin(MRR(E98A5,- 6B629))
FB118-00246 (    1) {} || ->  .                    origin(Induction(List[nil by B6DCB, cons by F647B]))
699A2-00247 ( 1000) {} || ->    butlast(cons(V2, V4)) = take(len(V4), cons(V2, V4)) .          origin(Derived(FB118
      ))
DF075-00248 (    1) {} || ->  .                    origin(Contradicting Set(61994) and Set(699A2)))
D618F-00249 (    1) {} || ->  .                    origin(Induction(List[nil by A2213, cons by DF075]))
8944B-00250 ( 1000) {} || ->    butlast(V2) = take(minus(len(V2), s(z)), V2) .        origin(Derived(D618F))
D8170-00251 (    1) {} || ->  .                    origin(Contradicting Set(FE538) and Set(8944B)))
Involved clauses: 38

        Total time: 2.551100824 s
```

A.1.2. Problem 85

Input

The problem in DFG format, edited for better readability.

```
begin_problem(benchmark).

list_of_descriptions.
name({**}).
author({**}).
status(unknown).
description({**}).
end_of_list.


list_of_symbols.
functions [(nil, 1+0), (cons, 1+2), (p,2+2), (z,0), (s,1),
           (append, 1+2), (rev, 1+1), (zip, 1+2), (len,1)].
predicates [(less, 1+2)].
end_of_list.

list_of_declarations.
function(nil, [A], list(A)).
function(cons, [A], (A,list(A)) list(A)).
function(p, [A,B], (A,B) pair(A,B)).

function(z, nat).
function(s, (nat) nat).

function(append, [A], (list(A),list(A)) list(A)).
function(rev, [A], (list(A)) list(A)).
function(zip, [A,B], (list(A),list(B)) list(pair(A,B))).
function(len, [A], (list(A)) nat).

predicate(less, [A], A, A).

datatype(pair, p).
datatype(nat, z, s).
datatype(list, nil, cons).
end_of_list.

list_of_formulae(axioms).
formula(forall_sorts([A], forall([LS:list(A)],
        equal:lr(append<A>(nil<A>,LS),LS))),      append_nil).
formula(forall_sorts([A], forall([LS:list(A), Y:A, YS:list(A)],
        equal:lr(append<A>(cons<A>(Y,YS),LS),
                 cons<A>(Y,append<A>(YS,LS))))), append_cons).

formula(forall_sorts([A],
        equal:lr(rev<A>(nil<A>),nil<A>)),          rev_nil).
formula(forall_sorts([A], forall([Y:A, YS:list(A)],
        equal:lr(rev<A>(cons<A>(Y,YS)),
                 append<A>(rev<A>(YS),cons<A>(Y,nil<A>))))),
                                                   rev_cons).

formula(forall_sorts([A],
        equal:lr(len<A>(nil<A>), z)),              len_nil).
formula(forall_sorts([A], forall([L:A, LS:list(A)],
        equal:lr(len<A>(cons<A>(L,LS)),
                 s(len<A>(LS))))),                 len_cons).

formula(forall_sorts([A], forall([XS:list(A)],
        equal:lr(zip<A,A>(XS,nil<A>),
                 nil<pair(A,A)>))),                zip_nil_right).
formula(forall_sorts([A], forall([YS:list(A)],
        equal:lr(zip<A,A>(nil<A>,YS),
                 nil<pair(A,A)>))),                zip_nil_left).
formula(forall_sorts([A], forall([X:A, XS:list(A),
                                  Y:A, YS:list(A)],
        equal:lr(zip<A,A>(cons<A>(X,XS),cons<A>(Y,YS)),
              cons<pair(A,A)>(p<A,A>(X,Y),zip<A,A>(XS,YS))))),
                                                   zip_cons).
end_of_list.

list_of_formulae(conjectures).
formula(forall_sorts([A], forall([X:A, XS:list(A), Y:A, YS:list(A)],
        implies(equal(len<A>(XS),len<A>(YS)),
                equal(zip<A,A>(rev<A>(XS),rev<A>(YS)),
                      rev<pair(A,A)>(zip<A,A>(XS,YS)))))),
                                                   conj_prob85).
end_of_list.

end_problem.
```

Output

The boxed parts are the unedited output of Pirate (except for adding bold for keywords).

```
Starting
scala version 2.11.2
```

The problem contains two negated-conjecture clauses.

```
Initial Conjectures:
  : 7DD03-00012 (    1) {} ||    zip(rev(skf1), rev(skf2)) = rev(zip(skf1, skf2)) ->  .          origin(Conj(
        conj_prob85 from: Set(A720D)))
  : 8A2C7-00011 (    1) {} ||  ->        len(skf1) = len(skf2) .                origin(Conj(conj_prob85 from: Set(
        A720D)))
```

The proof search derives three global axioms (see the *Welcoming new knowledge* lines) and has already quite a few states (*CaseProofTask*).

```
@        313  @head(3 rank: 7 ): CaseProofTask-TID:8( parent TID:5, Initial)
@        517  @head(3 rank: 10 ): CaseProofTask-TID:4( parent TID:3, DeriveCase( ->   zip(V2, nil) = nil . ))
Welcoming new knowledge:
  : 71DCE-00201 ( 1000) {} ||  ->       zip(V2, nil) lr=> nil* .                 origin(Derived(A17A6))
@        534  @head(2 rank: 10 ): CaseProofTask-TID:6( parent TID:3, DeriveCase( ->   append(V2, nil) = V2 . ))
Welcoming new knowledge:
  : 90D36-00227 ( 1000) {} ||  ->       append(V2, nil) lr=> V2* .               origin(Derived(47A3E))
@        604  @head(1 rank: 10 ): CaseProofTask-TID:7( parent TID:3, DeriveCase( ->   zip(nil, V2) = nil . ))
Welcoming new knowledge:
  : 9A748-00230 ( 1000) {} ||  ->       zip(nil, V2) lr=> nil* .                 origin(Derived(81519))
@        619  @head(0 rank: 22 ): CaseProofTask-TID:9( parent TID:8, Case(cons of skf2))
@        873  @head(1 rank: 24 ): CaseProofTask-TID:12( parent TID:10, DeriveCase(Set( ->   zip(rev(skf1), rev(V2)) =
        rev(zip(skf1, V2)) . ,   len(skf1) = s(len(i1002)) ->   zip(rev(skf1), rev(V2)) = rev(zip(skf1, V2)) . )))
@        995  @head(1 rank: 42 ): CaseProofTask-TID:11( parent TID:9, Case(cons of skf1))
@       1382  @head(9 rank: 24 ): CaseProofTask-TID:18( parent TID:17, MultiDeriveCase( ->   len(V2) = len(rev(V2)) .
        ))
@       1433  @head(9 rank: 40 ): CaseProofTask-TID:14( parent TID:11, Case(nil of i1006))
@       1472  @head(8 rank: 44 ): CaseProofTask-TID:13( parent TID:12, Case(cons of skf8))
@       1624  @head(10 rank: 40 ): CaseProofTask-TID:32( parent TID:31, MultiDeriveCase(  len(skf1) = s(len(i1002))
      ->   zip(rev(skf1), rev(V2)) = rev(zip(skf1, V2)) . ))
@       1689  @head(10 rank: 40 ): CaseProofTask-TID:33( parent TID:10, DeriveCase(Set( ->   zip(rev(skf1), rev(V2))
      = rev(zip(skf1, V2)) . ,   len(skf1) = s(len(i1002)) ->   zip(rev(skf1), rev(V2)) = rev(zip(skf1, V2)) . )))
@       1789  @head(10 rank: 52 ): CaseProofTask-TID:20( parent TID:10, DeriveCase(Set(  len(i1006) = len(i1002) ->
         zip(append(V2, cons(i1005, nil)), append(V4, cons(i1001, nil))) = append(zip(V2, V4), cons(p(i1005, i1001),
       nil)) . , ->   zip(append(V2, cons(i1005, nil)), append(V4, cons(i1001, nil))) = append(zip(V2, V4), cons(p(
       i1005, i1001), nil)) . )))
@       2019  @head(11 rank: 50 ): CaseProofTask-TID:36( parent TID:20, Case(nil of skf20))
@       2131  @head(11 rank: 24 ): CaseProofTask-TID:38( parent TID:36, Case(cons of skf19))
@       2491  @head(13 rank: 24 ): CaseProofTask-TID:40( parent TID:39, Case(nil of i1006))
@       2545  @head(13 rank: 27 ): CaseProofTask-TID:45( parent TID:40, Case(nil of i1002))
@       2569  @head(12 rank: 32 ): CaseProofTask-TID:41( parent TID:39, Case(cons of i1006))
@       2683  @head(11 rank: 32 ): CaseProofTask-TID:44( parent TID:42, Case(cons of i1002))
@       2757  @head(10 rank: 52 ): CaseProofTask-TID:27( parent TID:24, MultiDeriveCase(  len(V2) = len(V4) ->   zip(
       append(V2, cons(i1005, nil)), append(V4, cons(i1001, nil))) = append(zip(V2, V4), cons(p(i1005, i1001), nil))
       . ))
@       2913  @head(11 rank: 45 ): CaseProofTask-TID:46( parent TID:27, Case(nil of skf10))
@       2944  @head(10 rank: 55 ): CaseProofTask-TID:29( parent TID:18, Case(cons of skf17))
@       3040  @head(12 rank: 56 ): CaseProofTask-TID:16( parent TID:10, DeriveCase(Set(  len(i1006) = len(i1002) ->
         zip(append(V2, cons(i1005, nil)), append(rev(i1002), cons(i1001, nil))) = append(zip(V2, rev(i1002)), cons(p
       (i1005, i1001), nil)) . , ->   zip(append(V2, cons(i1005, nil)), append(rev(i1002), cons(i1001, nil))) =
       append(zip(V2, rev(i1002)), cons(p(i1005, i1001), nil)) . )))
@       3314  @head(14 rank: 45 ): CaseProofTask-TID:52( parent TID:16, Case(nil of i1002))
@       3386  @head(13 rank: 56 ): CaseProofTask-TID:19( parent TID:17, MultiDeriveCase(  len(V2) = len(i1002) ->
       zip(append(V2, cons(i1005, nil)), append(rev(i1002), cons(i1001, nil))) = append(zip(V2, rev(i1002)), cons(p(
       i1005, i1001), nil)) . ))
@       3547  @head(15 rank: 35 ): CaseProofTask-TID:66( parent TID:65, MultiDeriveCase( ->   len(V2) = len(append(
       nil, V2)) . ))
@       3551  @head(14 rank: 56 ): CaseProofTask-TID:23( parent TID:21, MultiDeriveCase(  len(i1006) = len(V2) ->
       zip(append(rev(i1006), cons(i1005, nil)), append(V2, cons(i1001, nil))) = append(zip(rev(i1006), V2), cons(p(
       i1005, i1001), nil)) . ))
@       3722  @head(18 rank: 35 ): CaseProofTask-TID:78( parent TID:77, MultiDeriveCase( ->   len(V2) = len(append(V2
       , nil)) . ))
@       3725  @head(17 rank: 40 ): CaseProofTask-TID:75( parent TID:73, MultiDeriveCase( ->   len(V2) = len(append(
       nil, rev(V2))) . ))
@       3770  @head(16 rank: 45 ): CaseProofTask-TID:71( parent TID:23, Case(nil of i1006))
@       3803  @head(15 rank: 56 ): CaseProofTask-TID:28( parent TID:10, DeriveCase(Set( ->   zip(append(rev(i1006),
       cons(i1005, nil)), append(V2, cons(i1001, nil))) = append(zip(rev(i1006), V2), cons(p(i1005, i1001), nil)) . ,
          len(i1006) = len(i1002) ->   zip(append(rev(i1006), cons(i1005, nil)), append(V2, cons(i1001, nil))) =
       append(zip(rev(i1006), V2), cons(p(i1005, i1001), nil)) . )))
@       4098  @head(17 rank: 45 ): CaseProofTask-TID:83( parent TID:28, Case(nil of i1006))
@       4159  @head(17 rank: 18 ): CaseProofTask-TID:93( parent TID:83, Case(cons of skf21))
@       4210  @head(17 rank: 21 ): CaseProofTask-TID:94( parent TID:93, Case(nil of i1002))
@      4234  @head(16 rank: 60 ): CaseProofTask-TID:34( parent TID:32, Case(cons of skf1))
@      4339  @head(17 rank: 49 ): CaseProofTask-TID:96( parent TID:10, DeriveCase(Set( ->   zip(rev(V2), rev(skf23))
        = rev(zip(V2, skf23)) . ,   len(i1018) = len(i1002) ->   zip(rev(V2), rev(skf23)) = rev(zip(V2, skf23)) . )))
@      4402  @head(17 rank: 60 ): CaseProofTask-TID:35( parent TID:33, Case(cons of skf1))
@      4554  @head(18 rank: 49 ): CaseProofTask-TID:99( parent TID:10, DeriveCase(Set(  len(i1020) = len(i1002) ->
        zip(rev(V2), rev(skf26)) = rev(zip(V2, skf26)) . , ->   zip(rev(V2), rev(skf26)) = rev(zip(V2, skf26)) . ))
       )
@      4665  @head(18 rank: 60 ): CaseProofTask-TID:50( parent TID:10, DeriveCase( ->   s(len(V2)) = len(append(V2,
       cons(i1011, nil))) . ))
@      4699  @head(17 rank: 65 ): CaseProofTask-TID:58( parent TID:56, MultiDeriveCase(  len(i1006) = len(V2) ->
       zip(append(skf22, cons(i1005, nil)), append(V2, cons(i1001, nil))) = append(zip(skf22, V2), cons(p(i1005,
       i1001), nil)) . ))
@      4825  @head(17 rank: 65 ): CaseProofTask-TID:69( parent TID:10, DeriveCase(Set(  len(skf16) = len(i1002) ->
        zip(append(skf16, cons(i1005, nil)), append(V2, cons(i1001, nil))) = append(zip(skf16, V2), cons(p(i1005,
       i1001), nil)) . , ->  zip(append(skf16, cons(i1005, nil)), append(V2, cons(i1001, nil))) = append(zip(skf16,
        V2), cons(p(i1005, i1001), nil)) . )))
@      4993  @head(17 rank: 65 ): CaseProofTask-TID:74( parent TID:73, MultiDeriveCase(  len(V2) = len(skf13) ->
       zip(append(V2, cons(i1005, nil)), append(skf13, cons(i1001, nil))) = append(zip(V2, skf13), cons(p(i1005,
       i1001), nil)) . ))
@      5107  @head(17 rank: 65 ): CaseProofTask-TID:92( parent TID:90, MultiDeriveCase(  len(V2) = len(i1002) ->
       zip(append(V2, cons(i1005, nil)), append(skf21, cons(i1001, nil))) = append(zip(V2, skf21), cons(p(i1005,
       i1001), nil)) . ))
@      5230  @head(18 rank: 60 ): CaseProofTask-TID:107( parent TID:92, Case(nil of skf21))
@      5290  @head(18 rank: 28 ): CaseProofTask-TID:109( parent TID:107, Case(cons of skf56))
@      5346  @head(18 rank: 32 ): CaseProofTask-TID:110( parent TID:109, Case(cons of i1002))
@      5511  @head(20 rank: 30 ): CaseProofTask-TID:112( parent TID:111, Case(nil of i1078))
@      5545  @head(20 rank: 33 ): CaseProofTask-TID:117( parent TID:112, Case(nil of i1080))
@      5560  @head(19 rank: 40 ): CaseProofTask-TID:113( parent TID:111, Case(cons of i1078))
@      5597  @head(18 rank: 40 ): CaseProofTask-TID:116( parent TID:114, Case(cons of i1080))
@      5638  @head(17 rank: 70 ): CaseProofTask-TID:30( parent TID:13, Case(cons of skf1))
@      5938  @head(24 rank: 60 ): CaseProofTask-TID:118( parent TID:30, Case(nil of i1016))
@      6027  @head(24 rank: 72 ): CaseProofTask-TID:49( parent TID:10, DeriveCase( ->   s(len(rev(i1012))) = len(
       append(rev(i1012), cons(i1011, nil))) . ))
@      6028  @head(23 rank: 75 ): CaseProofTask-TID:37( parent TID:20, Case(cons of skf20))
@      6703  @head(30 rank: 28 ): CaseProofTask-TID:134( parent TID:133, Case(nil of skf19))
@      7054  @head(31 rank: 36 ): CaseProofTask-TID:149( parent TID:147, Case(cons of i1006))
@      7153  @head(30 rank: 36 ): CaseProofTask-TID:152( parent TID:150, Case(cons of i1002))
@      7234  @head(29 rank: 75 ): CaseProofTask-TID:47( parent TID:27, Case(cons of skf10))
```

```
Found false
Formulas used:  len_cons conj_prob85 rev_cons Generalization-with-side-condition-B91EE len_nil Side-Cond-B91EE
       append_nil rev_nil zip_cons Generalization-60BD0 zip_nil_right append_cons
```

```
Involved clauses:
A720D-00001 ( 1000) {} ||       len(V2) = len(V4) ->    zip(rev(V2), rev(V4)) = rev(zip(V2, V4)) .
       origin(Input(conj_prob85))
77D48-00002 ( 1000) {} || ->   append(nil, V2) = V2 .          origin(Input(append_nil))
A6F4B-00003 ( 1000) {} || ->   append(cons(V2, V4), V6) = cons(V2, append(V4, V6)) .          origin(Input(
       append_cons))
4622A-00004 ( 1000) {} || ->   rev(nil) = nil .          origin(Input(rev_nil))
745AD-00005 ( 1000) {} || ->   rev(cons(V2, V4)) = append(rev(V4), cons(V2, nil)) .          origin(Input(
       rev_cons))
B7230-00006 ( 1000) {} || ->   len(nil) = z .          origin(Input(len_nil))
E1EC9-00007 ( 1000) {} || ->   len(cons(V2, V4)) = s(len(V4)) .          origin(Input(len_cons))
71DCE-00008 ( 1000) {} || ->   zip(V2, nil) = nil .          origin(Input(zip_nil_right))
51B06-00010 ( 1000) {} || ->   zip(cons(V2, V4), cons(V6, V8)) = cons(p(V2, V6), zip(V4, V8)) .
       origin(Input(zip_cons))
8A2C7-00011 (    1) {} || ->   len(skf1) = len(skf2) .          origin(Conj(conj_prob85 from: Set(A720D)))
7DD03-00012 (    1) {} ||      zip(rev(skf1), rev(skf2)) = rev(zip(skf1, skf2)) -> .          origin(Conj(
       conj_prob85 from: Set(A720D)))
```

```
36DB6-00167 (    1) {} ||      zip(rev(skf1), rev(nil)) = rev(zip(skf1, nil)) -> .          origin(IndCase(nil,
       skf2, 7DD03))
A5244-00170 (    1) {} ||      nil = nil -> .          origin(Rew(36DB6 with Set[71DCE, 4622A]))
DEAD1-00171 (    1) {} || -> .          origin(LitElimF(A5244))
A4424-00172 (    1) {} ||      zip(rev(skf1), rev(cons(i1001, i1002))) = rev(zip(skf1, cons(i1001, i1002))) -> .
                origin(IndCase(cons, skf2, 7DD03))
CC891-00173 (    1) {} || ->   len(skf1) = len(cons(i1001, i1002)) .          origin(Rew(CC891 with Set[E1EC9]))
F27CB-00176 (    1) {} ||      zip(rev(skf1), append(rev(i1002), cons(i1001, nil))) = rev(zip(skf1, cons(i1001,
       i1002))) -> .          origin(Rew(A4424 with Set[745AD]))
FCB76-00177 (    1) {} || ->   len(skf1) = s(len(i1002)) .          origin(Rew(CC891 with Set[E1EC9]))
8F158-00307 (    1) {} || ->   len(nil) = s(len(i1002)) .          origin(IndCase(nil, skf1, FCB76))
84A5B-00311 (    1) {} || ->   z = s(len(i1002)) .          origin(Rew(8F158 with Set[B7230]))
CD288-00312 (    1) {} || -> .          origin(DT-Distinct(84A5B))
3FE76-00313 (    1) {} || ->   len(cons(i1005, i1006)) = s(len(i1002)) .          origin(IndCase(cons, skf1,
       FCB76))
97C61-00314 (    1) {} ||      zip(rev(cons(i1005, i1006)), append(rev(i1002), cons(i1001, nil))) = rev(zip(cons(
       i1005, i1006), cons(i1001, i1002))) -> .          origin(IndCase(cons, skf1, F27CB))
ED4BD-00325 (    1) {} || ->   s(len(i1006)) = s(len(i1002)) .          origin(Rew(3FE76 with Set[E1EC9]))
B91EE-00326 (    1) {} ||      zip(append(rev(i1006), cons(i1005, nil)), append(rev(i1002), cons(i1001, nil))) =
       append(rev(zip(i1006, i1002)), cons(p(i1005, i1001), nil)) -> .          origin(Rew(97C61 with Set[51B06,
        745AD]))
```

```
09165-00331 (     1) {} ||  ->   len(i1006) = len(i1002) .              origin(DT-Injective(ED4BD))
DFF55-00915 (     1) {} ||       len(V2) = len(V4) ->     zip(append(V2, cons(i1005, nil)), append(V4, cons(i1001, nil
      ))) = append(zip(V2, V4), cons(p(i1005, i1001), nil)) .              origin(Input(non neg conj))
7E880-00916 (     1) {} ||  ->   len(skf9) = len(skf10) .               origin(Conj(Generalization-with-side-
      condition-B91EE from: Set(DFF55)))
FA5C4-00917 (     1) {} ||       zip(append(skf9, cons(i1005, nil)), append(skf10, cons(i1001, nil))) = append(zip(
      skf9, skf10), cons(p(i1005, i1001), nil)) ->  .               origin(Conj(Generalization-with-side-condition-
      B91EE from: Set(DFF55)))
0ECF8-01046 (     1) {} ||  ->   len(V2) = len(rev(V2)) .               origin(Input(Side-Cond-non-Neg-B91EE))
14F20-01047 (     1) {} ||       len(skf17) = len(rev(skf17)) ->  .              origin(Conj(Side-Cond-B91EE from:
      Set(0ECF8)))
64F41-01219 (     1) {} ||       len(nil) = len(rev(nil)) ->  .         origin(IndCase(nil, skf17, 14F20))
2F5DE-01220 (     1) {} ||       z = z ->  .                 origin(Rew(64F41 with Set[B7230, 4622A]))
756DD-01221 (     1) {} ||  ->  .                  origin(LitElimF(2F5DE))
60BD0-01225 (     1) {} ||       s(len(i1012)) = len(append(rev(i1012), cons(i1011, nil))) ->  .
      origin(Rew(2D869 with Set[E1EC9, 745AD]))
C5074-03855 (     1) {} ||       len(skf9) = len(skf10) ->      zip(append(skf9, cons(i1005, nil)), append(skf10,
      cons(i1001, nil))) = append(zip(skf9, skf10), cons(p(i1005, i1001), nil)) .          origin(Negate(Set(FA5C4,
       7E880)))
91218-03860 (     1) {} ||       zip(append(skf9, cons(i1005, nil)), append(nil, cons(i1001, nil))) = append(zip(skf9
      , nil), cons(p(i1005, i1001), nil)) ->  .             origin(IndCase(nil, skf10, FA5C4))
DBF5A-03861 (     1) {} ||  ->   len(skf9) = len(nil) .            origin(IndCase(nil, skf10, 7E880))
5D4D3-03862 (     1) {} ||       zip(append(skf9, cons(i1005, nil)), cons(i1001, nil)) = cons(p(i1005, i1001), nil)
     ->  .               origin(Rew(91218 with Set[71DCE, 77D48]))
226BF-03863 (     1) {} ||  ->   len(skf9) = z .                   origin(Rew(DBF5A with Set[B7230]))
D0306-03920 (     1) {} ||       zip(append(skf9, cons(i1005, nil)), append(cons(i1035, i1036), cons(i1001, nil))) =
      append(zip(skf9, cons(i1035, i1036)), cons(p(i1005, i1001), nil)) ->  .          origin(IndCase(cons, skf10
      , FA5C4))
D35CD-03921 (     1) {} ||  ->   len(skf9) = len(cons(i1035, i1036)) .           origin(IndCase(cons, skf10, 7E880))
B68C6-03923 (     1) {} ||       len(skf9) = len(i1036) ->      zip(append(skf9, cons(i1005, nil)), append(i1036,
      cons(i1001, nil))) = append(zip(skf9, i1036), cons(p(i1005, i1001), nil)) .          origin(IndHyp(skf10,
      cons: C5074))
66512-03924 (     1) {} ||  ->   len(skf9) = s(len(i1036)) .            origin(Rew(D35CD with Set[E1EC9]))
7E53F-03925 (     1) {} ||       zip(append(skf9, cons(i1005, nil)), cons(i1035, append(i1036, cons(i1001, nil)))) =
      append(zip(skf9, cons(i1035, i1036)), cons(p(i1005, i1001), nil)) ->  .          origin(Rew(D0306 with Set[
      A6F4B]))
35A58-04058 (     1) {} ||       zip(append(nil, cons(i1005, nil)), cons(i1001, nil)) = cons(p(i1005, i1001), nil) ->
       .              origin(IndCase(nil, skf9, 5D4D3))
0A341-04059 (     1) {} ||       cons(p(i1005, i1001), zip(nil, nil)) = cons(p(i1005, i1001), nil) ->  .
           origin(Rew(35A58 with Set[51B06, 77D48]))
C42E9-04060 (     1) {} ||       zip(nil, nil) = nil     p(i1005, i1001) = p(i1005, i1001) ->  .
      origin(DT-Injective(0A341))
1B501-04061 (     1) {} ||       zip(nil, nil) = nil ->  .              origin(LitElim(C42E9))
6778B-04062 (     1) {} ||  ->  .                  origin(MRR(1B501,- 71DCE))
FE9E8-04063 (     1) {} ||  ->   len(cons(i1037, i1038)) = z .          origin(IndCase(cons, skf9, 226BF))
90922-04068 (     1) {} ||  ->   s(len(i1038)) = z .               origin(Rew(FE9E8 with Set[E1EC9]))
A7CD7-04069 (     1) {} ||  ->  .                  origin(DT-Distinct(90922))
142A0-04070 (     1) {} ||  ->  .                  origin(Induction(List[nil by 6778B, cons by A7CD7]))
5CD72-04071 ( 1000) {} ||       z = len(V2) ->         zip(append(V2, cons(V4, nil)), cons(V6, nil)) = cons(p(V4,
      V6), nil) .             origin(Derived(142A0))
E97D7-04072 (     1) {} ||  ->  .                  origin(Contradicting Set(5D4D3, 226BF) and Set(5CD72)))
41691-04265 (     1) {} ||       s(len(skf28)) = len(append(skf28, cons(i1011, nil))) ->  .              origin(Conj(
      Generalization-60BD0 from: Set(60BD0)))
0F1FF-07739 (     1) {} ||  ->   s(len(skf28)) = len(append(skf28, cons(i1011, nil))) .          origin(Negate(Set
      (41691)))
71C49-07742 (     1) {} ||       s(len(nil)) = len(append(nil, cons(i1011, nil))) ->  .          origin(IndCase(nil,
      skf28, 41691))
346BC-07745 (     1) {} ||       s(z) = s(len(nil)) ->  .              origin(Rew(71C49 with Set[B7230, E1EC9, 77
      D48]))
D6A23-07746 (     1) {} ||       len(nil) = z ->  .             origin(DT-Injective(346BC))
F9091-07747 (     1) {} ||  ->  .                  origin(MRR(D6A23,- B7230))
E6401-07748 (     1) {} ||       s(len(cons(i1067, i1068))) = len(append(cons(i1067, i1068), cons(i1011, nil))) ->  .
            origin(IndCase(cons, skf28, 41691))
B593D-07755 (     1) {} ||  ->   s(len(i1068)) = len(append(i1068, cons(i1011, nil))) .          origin(IndHyp(skf28,
       cons: 0F1FF))
6DB94-07757 (     1) {} ||       s(s(len(i1068))) = s(len(append(i1068, cons(i1011, nil)))) ->  .
      origin(Rew(E6401 with Set[A6F4B, E1EC9]))
3E49B-07758 (     1) {} ||       s(len(i1068)) = len(append(i1068, cons(i1011, nil))) ->  .              origin(DT-
      Injective(6DB94))
23DC0-07759 (     1) {} ||  ->  .                  origin(MRR(3E49B,- B593D))
F6F97-07760 (     1) {} ||  ->  .                  origin(Induction(List[nil by F9091, cons by 23DC0]))
6BC4C-07763 (     1) {} ||  ->   s(len(V2)) = len(append(V2, cons(i1011, nil))) .              origin(Derived(F6F97
      ))
032E0-07764 (     1) {} ||       s(len(skf28)) = s(len(skf28)) ->  .          origin(Rew(41691 with Set[6BC4C]))
368A6-07765 (     1) {} ||  ->  .                  origin(LitElimF(032E0))
1BC46-07766 (     1) {} ||  ->  .                  origin(Induction(List[nil by 756DD, cons by 368A6]))
0ECF8-07767 ( 1000) {} ||  ->   len(V2) = len(rev(V2)) .               origin(Derived(1BC46))
062F6-13532 (     1) {} ||  ->   len(nil) = s(len(i1036)) .             origin(IndCase(nil, skf9, 66512))
E2ECB-13538 (     1) {} ||  ->   z = s(len(i1036)) .               origin(Rew(062F6 with Set[B7230]))
BBFD0-13539 (     1) {} ||  ->  .                  origin(DT-Distinct(E2ECB))
D0964-13540 (     1) {} ||  ->   len(cons(i1111, i1112)) = s(len(i1036)) .           origin(IndCase(cons, skf9,
      66512))
B1C55-13541 (     1) {} ||       zip(append(cons(i1111, i1112), cons(i1005, nil)), cons(i1035, append(i1036, cons(
      i1001, nil)))) = append(zip(cons(i1111, i1112), cons(i1035, i1036)), cons(p(i1005, i1001), nil)) ->  .
```

```
                    origin(IndCase(cons, skf9, 7E53F))
4E6E4-13543 (    1) {} ||       len(i1112) = len(i1036) ->      zip(append(i1112, cons(i1005, nil)), append(i1036,
      cons(i1001, nil))) = append(zip(i1112, i1036), cons(p(i1005, i1001), nil)) .              origin(IndHyp(
      skf9, cons: B68C6))
C2FB1-13552 (    1) {} ||       cons(p(i1111, i1035), zip(append(i1112, cons(i1005, nil)), append(i1036, cons(i1001,
       nil)))) = cons(p(i1111, i1035), append(zip(i1112, i1036), cons(p(i1005, i1001), nil))) -> .
      origin(Rew(B1C55 with Set[A6F4B, 51B06]))
4C358-13553 (    1) {} || ->    s(len(i1112)) = s(len(i1036)) .                origin(Rew(D0964 with Set[E1EC9]))
05196-13558 (    1) {} || ->    len(i1112) = len(i1036) .               origin(DT-Injective(4C358))
C1ECD-13559 (    1) {} ||       zip(append(i1112, cons(i1005, nil)), append(i1036, cons(i1001, nil))) = append(zip(
      i1112, i1036), cons(p(i1005, i1001), nil))   p(i1111, i1035) = p(i1111, i1035) -> .          origin(DT
      -Injective(C2FB1))
4375A-13560 (    1) {} ||       zip(append(i1112, cons(i1005, nil)), append(i1036, cons(i1001, nil))) = append(zip(
      i1112, i1036), cons(p(i1005, i1001), nil)) -> .           origin(LitElim(C1ECD))
A0487-13562 (    1) {} || ->    zip(append(i1112, cons(i1005, nil)), append(i1036, cons(i1001, nil))) = append(zip(
      i1112, i1036), cons(p(i1005, i1001), nil)) .              origin(MRR(05196,- 4E6E4))
724C8-13567 (    1) {} || -> .                     origin(MRR(A0487,- 4375A))
CFF69-13568 (    1) {} || -> .                     origin(Induction(List[nil by BBFD0, cons by 724C8]))
C1C89-13569 ( 1000) {} ||       len(V2) = s(len(V4)) ->         zip(append(V2, cons(V6, nil)), cons(V8, append(V4,
      cons(V10, nil)))) = append(zip(V2, cons(V8, V4)), cons(p(V6, V10), nil)) .       origin(Derived(CFF69))
AB7BC-13570 (    1) {} || -> .                     origin(Contradicting Set(7E53F, 66512) and Set(C1C89)))
35234-13571 (    1) {} || -> .                     origin(Induction(List[nil by E97D7, cons by AB7BC]))
424F8-13572 ( 1000) {} ||       len(V2) = len(V4) ->    zip(append(V2, cons(V6, nil)), append(V4, cons(V8, nil))) =
      append(zip(V2, V4), cons(p(V6, V8), nil)) .              origin(Derived(35234))
A2C66-13575 (    1) {} || -> .                     origin(Contradicting Set(B91EE, 09165) and Set(424F8, 0ECF8)))
ABE6D-13576 (    1) {} || -> .                     origin(Induction(List[nil by CD288, cons by A2C66]))
8991D-13577 ( 1000) {} ||       len(V2) = s(len(V4)) ->         zip(rev(V2), append(rev(V4), cons(V6, nil))) = rev(
      zip(V2, cons(V6, V4))) .             origin(Derived(ABE6D))
CE2A4-13578 (    1) {} || -> .                     origin(Contradicting Set(F27CB, FCB76) and Set(8991D)))
38F66-13579 (    1) {} || -> .                     origin(Induction(List[nil by DEAD1, cons by CE2A4]))
A720D-13580 ( 1000) {} ||       len(V2) = len(V4) ->    zip(rev(V2), rev(V4)) = rev(zip(V2, V4)) .
      origin(Derived(38F66))
137DF-13581 (    1) {} || -> .                     origin(Contradicting Set(7DD03, 8A2C7) and Set(A720D)))
Involved clauses: 98

      Total time: 8.607543767 s
```

A.1.3. Problem 86

Input

The problem in DFG format, edited for better readability.

```
begin_problem(benchmark).

list_of_descriptions.
name({**}).
author({**}).
status(unknown).
description({**}).
end_of_list.


list_of_symbols.
functions [(z,0), (s,1), (cons, 1+2), (nil, 1+0), (ins,1+2)].
predicates [(less,1+2), (elem,1+2)].
sorts [(list,1),(nat,0)].
end_of_list.

list_of_declarations.
function(z, nat).
function(s, (nat) nat).

function(nil,    [A], list(A)).
function(cons,   [A], (A, list(A)) list(A)).

function(ins,    [A], (A, list(A)) list(A)).

predicate(less, [A], A, A).
predicate(elem, [A], A, list(A)).

datatype(nat, z, s).
datatype(list, nil, cons).
end_of_list.

list_of_formulae(axioms).
formula(forall([X:nat], not(less<nat>(X,z))),                   less_z_right).
formula(forall([X:nat], less<nat>(z,s(X))),                     less_z_left).
formula(forall([Y:nat, X:nat], equiv:lr(less<nat>(s(X),s(Y)),
                                  less<nat>(X,Y))),             less_z).

formula(forall_sorts([A], forall([X:A, LS:list(A)],
        not(elem<A>(X,nil<A>))))),                              elem_nil).
formula(forall_sorts([A], forall([X:A, L:A, LS:list(A)],
        implies(equal:lr(X, L),
                elem<A>(X,cons<A>(L,LS)))))),                   elem_cons_equal).
formula(forall_sorts([A], forall([X:A, L:A, LS:list(A)],
        implies(not(equal:lr(X, L)),
                equiv:lr(elem<A>(X,cons<A>(L,LS)),
                    elem<A>(X,LS)))))),                         elem_cons_notequal).

formula(forall([X:nat], equal:lr(ins<nat>(X,nil<nat>),
                            cons<nat>(X,nil<nat>)))),           ins_nil).
formula(forall([X:nat, Y:nat, YS:list(nat)],
        implies(less<nat>(X,Y),
                equal:lr(ins<nat>(X,cons<nat>(Y,YS)),
                    cons<nat>(X,cons<nat>(Y,YS)))))),           ins_less).
formula(forall([X:nat, Y:nat, YS:list(nat)],
        implies(not(less<nat>(X,Y)),
                equal:lr(ins<nat>(X,cons<nat>(Y,YS)),
                    cons<nat>(Y,ins<nat>(X,YS)))))),            ins_not_less).
end_of_list.


list_of_formulae(conjectures).
formula(forall([X:nat, Y:nat, LS:list(nat)],
        implies(less<nat>(X,Y),
                equiv(elem<nat>(X,ins<nat>(Y,LS)),
                    elem<nat>(X,LS))))),                        conj_86).
end_of_list.

end_problem.
```

Output

The boxed parts are the unedited output of Pirate (except for adding bold for keywords).

```
Starting
scala version 2.11.2
Unkown/Legacy-style Argument: /home/dwand/pirate/reduceDFGtoFacts
Initial Conjectures:
  : 917BF-00015 (    1) {} || ->       elem(skf1, ins(skf2, skf3))     elem(skf1, skf3) .          origin(Conj(
        conj_86 from: Set(6E1FE, 71276)))
  : 46111-00016 (    1) {} ||   elem(skf1, ins(skf2, skf3))     elem(skf1, skf3) ->  .          origin(Conj(conj_86
        from: Set(6E1FE, 71276)))
  : 78F19-00014 (    1) {} || ->       less(skf1, skf2) .              origin(Conj(conj_86 from: Set(6E1FE, 71276))
        )
```

```
@      299  @head(1 rank: 2 ): CaseProofTask-TID:4( parent TID:3, DeriveCase( ->   less(V2, V4)  less(V4, V2) . ))
@      778  @head(2 rank: 3 ): CaseProofTask-TID:6( parent TID:5, Initial)
@      983  @head(2 rank: 4 ): CaseProofTask-TID:8( parent TID:7, Case(z of skf4))
@     1061  @head(2 rank: 8 ): CaseProofTask-TID:9( parent TID:7, Case(s of skf4))
@       1142  @head(1 rank: 8 ): CaseProofTask-TID:14( parent TID:6, Case(s of skf2))
@       2606  @head(2 rank: 6 ): CaseProofTask-TID:16( parent TID:14, Case(nil of skf3))
@       2662  @head(1 rank: 15 ): CaseProofTask-TID:15( parent TID:8, Case(z of skf5))
@       2687  @head(0 rank: 15 ): CaseProofTask-TID:17( parent TID:14, Case(cons of skf3))
@       6852  @head(2 rank: 20 ): CaseProofTask-TID:20( parent TID:19, SplitCase(1))
@       7164  @head(1 rank: 20 ): CaseProofTask-TID:21( parent TID:19, SplitCase(2))
```

```
Found false
Formulas used:  ins_nil split-ground-pos elem_cons_notequal split-ground-neg elem_cons_equal elem_nil less_z_right
      ins_not_less ins_less less_z conj_86
```

```
Involved clauses:
6E1FE-00001 ( 1000) {} ||       elem(V2, ins(V4, V6))   less(V2, V4) ->         elem(V2, V6) .          origin(Input
      (conj_86))
71276-00002 ( 1000) {} ||       elem(V2, V4)    less(V2, V6) ->         elem(V2, ins(V6, V4)) .
      origin(Input(conj_86))
0D298-00003 ( 1000) {} ||       less(V2, z) ->  .               origin(Input(less_z_right))
E1868-00005 ( 1000) {} ||       less(s(V2), s(V4)) ->   less(V2, V4) .          origin(Input(less_z))
74758-00006 ( 1000) {} ||       less(V2, V4) ->         less(s(V2), s(V4)) .            origin(Input(less_z))
511EF-00007 ( 1000) {} ||       elem(V2, nil) ->  .             origin(Input(elem_nil))
59740-00008 ( 1000) {} ||       V2 = V4 ->      elem(V2, cons(V4, V6)) .                origin(Input(elem_cons_equal
      ))
F0650-00009 ( 1000) {} ||       elem(V2, cons(V4, V6)) ->       elem(V2, V6)    V2 = V4 .               origin(Input
      (elem_cons_notequal))
5C661-00010 ( 1000) {} ||       elem(V2, V4) ->         elem(V2, cons(V6, V4))  V2 = V6 .               origin(Input
      (elem_cons_notequal))
5C1C9-00011 ( 1000) {} ||  ->   ins(V2, nil) = cons(V2, nil) .          origin(Input(ins_nil))
8B37E-00012 ( 1000) {} ||       less(V2, V4) ->         ins(V2, cons(V4, V6)) = cons(V2, cons(V4, V6)) .
            origin(Input(ins_less))
7B2F6-00013 ( 1000) {} ||  ->   less(V2, V4)    ins(V2, cons(V4, V6)) = cons(V4, ins(V2, V6)) .
      origin(Input(ins_not_less))
78F19-00014 (    1) {} ||  ->   less(skf1, skf2) .              origin(Conj(conj_86 from: Set(6E1FE, 71276)))
917BF-00015 (    1) {} ||  ->   elem(skf1, ins(skf2, skf3))     elem(skf1, skf3) .              origin(Conj(conj_86
      from: Set(6E1FE, 71276)))
46111-00016 (    1) {} ||       elem(skf1, ins(skf2, skf3))     elem(skf1, skf3) ->  .          origin(Conj(conj_86
      from: Set(6E1FE, 71276)))
A507B-00019 ( 1000) {} ||       V2 = V2 ->      elem(V2, cons(V2, V4)) .                origin(NuV(59740))
B9A26-00020 ( 1000) {} ||  ->   elem(V2, cons(V2, V4)) .                origin(LitElim(A507B))
BA31D-00836 (    1) {} ||  ->   less(skf1, z) .                 origin(IndCase(z, skf2, 78F19))
C9099-00842 ( 1000) {} ||  ->   .                       origin(MRR(0D298,- BA31D))
27412-00843 (    1) {} ||  ->   elem(skf1, ins(s(i1003), skf3))         elem(skf1, skf3) .              origin(
      IndCase(s, skf2, 917BF))
F25C3-00844 (    1) {} ||  ->   less(skf1, s(i1003)) .          origin(IndCase(s, skf2, 78F19))
87558-00845 (    1) {} ||       elem(skf1, ins(s(i1003), skf3))         elem(skf1, skf3) ->  .          origin(
      IndCase(s, skf2, 46111))
D7328-01762 (    1) {} ||       elem(skf1, ins(s(i1003), skf3))         less(skf1, s(i1003)) ->         elem(skf1,
      skf3) .           origin(Negate(Set(F25C3, 87558, 27412)))
2A820-01763 (    1) {} ||       elem(skf1, skf3)        less(skf1, s(i1003)) ->         elem(skf1, ins(s(i1003),
      skf3)) .          origin(Negate(Set(F25C3, 87558, 27412)))
C3247-01777 (    1) {} ||  ->   elem(skf1, ins(s(i1003), nil))  elem(skf1, nil) .               origin(IndCase(nil,
      skf3, 27412))
F25C3-01779 (    1) {} ||  ->   less(skf1, s(i1003)) .          origin(IndCase(nil, skf3, F25C3))
BCC96-01786 (    1) {} ||  ->   elem(skf1, cons(s(i1003), nil))         elem(skf1, nil) .               origin(Rew(
      C3247 with Set[5C1C9]))
7D520-01821 (    1) {} ||  ->   elem(skf1, cons(s(i1003), nil)) .               origin(MRR(BCC96,- 511EF))
EC543-01841 (    2) {} ||  ->   elem(skf1, nil)         skf1 = s(i1003) .               origin(Res(7D520,- F0650))
315F8-01844 (    2) {} ||  ->   skf1 = s(i1003) .               origin(MRR(EC543,- 511EF))
88A75-01898 (    1) {} ||  ->   less(skf1, skf1) .              origin(Rew(F25C3 with Set[315F8]))
FED48-02373 (    1) {} ||  ->   elem(skf1, ins(s(i1003), cons(i1006, i1007)))   elem(skf1, cons(i1006, i1007)) .
                origin(IndCase(cons, skf3, 27412))
0FE6E-02374 (    1) {} ||       elem(skf1, ins(s(i1003), cons(i1006, i1007)))   elem(skf1, cons(i1006, i1007)) ->  .
              origin(IndCase(cons, skf3, 87558))
F25C3-02375 (    1) {} ||  ->   less(skf1, s(i1003)) .          origin(IndCase(cons, skf3, F25C3))
BB79B-02387 (    1) {} ||       elem(skf1, ins(s(i1003), i1007))        less(skf1, s(i1003)) ->         elem(skf1,
      i1007) .          origin(IndHyp(skf3, cons: D7328))
ECBDB-02391 (    1) {} ||       elem(skf1, i1007)       less(skf1, s(i1003)) ->         elem(skf1, ins(s(i1003),
      i1007)) .         origin(IndHyp(skf3, cons: 2A820))
3C405-02398 (    1) {} ||       elem(skf1, ins(s(i1003), i1007)) ->     elem(skf1, i1007) .             origin(MRR(
      BB79B,- F25C3))
F08A5-02399 (    1) {} ||       elem(skf1, i1007) ->    elem(skf1, ins(s(i1003), i1007)) .              origin(MRR(
      ECBDB,- F25C3))
25904-05953 (    1) {} ||       elem(V2, cons(V2, V4))  less(skf1, skf1) ->     elem(V2, nil) .
      origin(Negate(Set(511EF, B9A26, 88A75)))
17406-05956 (    1) {} ||  ->   less(z, z) .            origin(IndCase(z, skf1, 88A75))
2C5ED-05965 (    1) {} ||  ->   .               origin(MRR(17406,- 0D298))
511EF-05966 ( 1000) {} ||       elem(V2, nil) ->  .             origin(IndCase(s, skf1, 511EF))
44697-05967 (    1) {} ||  ->   less(s(i1008), s(i1008)) .              origin(IndCase(s, skf1, 88A75))
B9A26-05968 ( 1000) {} ||  ->   elem(V2, cons(V2, V4)) .                origin(IndCase(s, skf1, B9A26))
A05AC-05978 (    1) {} ||       elem(V2, cons(V2, V4))  less(i1008, i1008) ->   elem(V2, nil) .
      origin(IndHyp(skf1, s: 25904))
6A6BD-05981 (    1) {} ||       less(i1008, i1008) ->   elem(V2, nil) .                 origin(MRR(A05AC,- B9A26))
5B612-06010 (    1) {} ||  ->   less(i1008, i1008) .            origin(IPSC(44697 by E1868 and 74758))
```

```
D552C-06015 (    1) {} || ->  elem(V2, nil) .                origin(MRR(6A6BD,- 5B612))
36F33-06017 (    1) {} || ->  .                 origin(MRR(D552C,- 511EF))
05200-06018 (    1) {} || ->  .                 origin(Induction(List[z by 2C5ED, s by 36F33]))
E3E4C-06019 ( 1000) {} ||      elem(V2, cons(V2, V4))  less(V6, V6) ->        elem(V2, nil) .
      origin(Derived(05200))
39EE5-06020 (    1) {} || ->  .                 origin(Contradicting Set(511EF, B9A26, 88A75) and Set(E3E4C)))
00AE6-06196 (    2) {} ||      less(s(i1003), i1006) ->       elem(skf1, cons(s(i1003), cons(i1006, i1007)))  elem
      (skf1, cons(i1006, i1007)) .             origin(Sup(FED48 by 8B37E))
BBA64-06198 (    2) {} ||      elem(skf1, cons(s(i1003), cons(i1006, i1007)))  elem(skf1, cons(i1006, i1007))  less
      (s(i1003), i1006) ->  .            origin(Sup(0FE6E by 8B37E))
B288B-06419 (    2) {} || ->  elem(skf1, cons(i1006, ins(s(i1003), i1007)))   elem(skf1, cons(i1006, i1007))  less
      (s(i1003), i1006) .               origin(Sup(FED48 by 7B2F6))
4A957-06421 (    2) {} ||      elem(skf1, cons(i1006, ins(s(i1003), i1007)))   elem(skf1, cons(i1006, i1007)) ->
           less(s(i1003), i1006) .              origin(Sup(0FE6E by 7B2F6))
88FD4-06584 (    3) {} ||      elem(skf1, ins(s(i1003), i1007))       elem(skf1, cons(i1006, i1007)) ->       less
      (s(i1003), i1006)   skf1 = i1006 .          origin(Res(4A957,- 5C661))
5B066-06727 (    3) {} ||      elem(skf1, cons(i1006, i1007))  less(s(i1003), i1006) ->        skf1 = s(i1003) .
                  origin(Res(BBA64,- 5C661))
351CF-06840 (    3) {} || ->  elem(skf1, ins(s(i1003), i1007))       elem(skf1, cons(i1006, i1007))  less(s(i1003
      ), i1006)   skf1 = i1006 .        origin(Res(F0650,- B288B))
F0CF3-06981 (    3) {} ||      less(s(i1003), i1006) ->       elem(skf1, cons(i1006, i1007))  skf1 = s(i1003) .
                  origin(Res(F0650,- 00AE6))
553C2-07728 (    4) {} ||      less(s(i1003), i1006) ->       skf1 = s(i1003) .                 origin(Res(5B066,-
      F0CF3))
A62A0-07948 (    2) {} || ->  elem(skf1, cons(i1006, i1007))  elem(skf1, i1007)       less(s(i1003), i1006)   skf1
        = i1006 .        origin(Res(3C405,- 351CF))
A6F69-08122 (    3) {} || ->  elem(skf1, i1007)       less(s(i1003), i1006)   skf1 = i1006 .            origin(Res(
      F0650,- A62A0))
1ACCD-08463 (    2) {} ||      elem(skf1, cons(i1006, i1007))  elem(skf1, i1007) ->    less(s(i1003), i1006)   skf1
        = i1006 .        origin(Res(F08A5,- 88FD4))
FF5D8-08656 (    3) {} ||      elem(skf1, i1007) ->    less(s(i1003), i1006)   skf1 = i1006 .            origin(Res(5
      C661,- 1ACCD))
F3F30-24223 ( 1000) {} ||      less(s(i1003), i1006) ->  .             origin(Input(split-ground-neg))
5B46C-24224 ( 1000) {} || ->  ins(s(i1003), cons(i1006, i1007)) = cons(i1006, ins(s(i1003), i1007)) .
              origin(Res(F3F30,- 7B2F6))
C5321-24225 ( 1000) {} || ->  less(s(i1003), i1006) .              origin(Input(split-ground-pos))
93366-24226 ( 1000) {} || ->  ins(s(i1003), cons(i1006, i1007)) = cons(s(i1003), cons(i1006, i1007)) .
              origin(Res(C5321,- 8B37E))
80AF4-24335 (    1) {} ||      elem(skf1, cons(i1006, ins(s(i1003), i1007)))   elem(skf1, cons(i1006, i1007)) ->  .
                origin(Rew(0FE6E with Set[5B46C]))
3A9D5-24338 (    3) {} ||      elem(skf1, i1007) ->    skf1 = i1006 .           origin(MRR(FF5D8,- F3F30))
0C699-24357 (    3) {} || ->  elem(skf1, i1007)       skf1 = i1006 .           origin(MRR(A6F69,- F3F30))
5D37C-25023 (    4) {} || ->  skf1 = i1006 .          origin(Res(3A9D5,- 0C699))
23964-25073 (    1) {} ||      elem(skf1, cons(skf1, ins(s(i1003), i1007)))    elem(skf1, cons(skf1, i1007)) ->  .
                origin(Rew(80AF4 with Set[5D37C]))
8F0AA-25149 (    1) {} ||      elem(skf1, cons(skf1, i1007)) ->  .             origin(MRR(23964,- B9A26))
D7686-25185 (    1) {} || ->  .                 origin(MRR(8F0AA,- B9A26))
08B6D-25186 (    1) {} ||      elem(skf1, cons(s(i1003), cons(i1006, i1007)))  elem(skf1, cons(i1006, i1007)) ->  .
                origin(Rew(0FE6E with Set[93366]))
315F8-25198 (    4) {} || ->  skf1 = s(i1003) .                origin(MRR(553C2,- C5321))
88A75-25341 (    1) {} || ->  less(skf1, skf1) .               origin(SRew(F25C3 by 315F8))
7B2F8-25552 (    1) {} ||      elem(skf1, cons(skf1, cons(i1006, i1007)))      elem(skf1, cons(i1006, i1007)) ->  .
                origin(Rew(08B6D with Set[315F8]))
3F5D5-25628 (    1) {} ||      elem(skf1, cons(i1006, i1007)) ->  .            origin(MRR(7B2F8,- B9A26))
EE209-27108 (    1) {} ||      elem(V2, cons(V2, V4))  less(skf1, skf1) ->     elem(skf1, cons(i1006, i1007)) .
                   origin(Negate(Set(3F5D5, B9A26, 88A75)))
17406-27112 (    1) {} || ->  less(z, z) .            origin(IndCase(z, skf1, 88A75))
C3E76-27130 ( 1000) {} || ->  .                 origin(MRR(0D298,- 17406))
9D08C-27131 (    1) {} || ->  less(s(i1010), s(i1010)) .             origin(IndCase(s, skf1, 88A75))
B9A26-27133 ( 1000) {} || ->  elem(V2, cons(V2, V4)) .               origin(IndCase(s, skf1, B9A26))
C14A0-27151 (    1) {} ||      elem(V2, cons(V2, V4))  less(i1010, i1010) ->   elem(i1010, cons(i1006, i1007)) .
                  origin(IndHyp(skf1, s: EE209))
B7AE6-27156 (    1) {} ||      elem(i1010, cons(i1006, i1007)) ->  .           origin(IndHyp(skf1, s: 3F5D5))
03C29-27192 (    1) {} ||      elem(V2, cons(V2, V4))  less(i1010, i1010) ->  .                origin(MRR(C14A0,-
      B7AE6))
D1845-27249 (    1) {} || ->  less(i1010, i1010) .             origin(IPSC(9D08C by E1868 and 74758))
47D12-27255 (    1) {} ||      less(i1010, i1010) ->  .              origin(MRR(03C29,- B9A26))
4EC41-27257 (    1) {} || ->  .                 origin(MRR(47D12,- D1845))
B82D3-27258 (    1) {} || ->  .                 origin(Induction(List[z by C3E76, s by 4EC41]))
B4241-27259 ( 1000) {} ||      elem(V2, cons(V2, V4))  less(V6, V6) ->        elem(V6, cons(V8, V10)) .
              origin(Derived(B82D3))
04B82-27260 (    1) {} || ->  .                 origin(Contradicting Set(3F5D5, B9A26, 88A75) and Set(B4241)))
C58A8-27261 (    1) {} || ->  .                 origin(BySplit(D7686,- 04B82))
CCC35-27262 (    1) {} || ->  .                 origin(Induction(List[nil by 39EE5, cons by C58A8]))
DBE1F-27263 ( 1000) {} ||      elem(V2, V4)    less(V2, s(V6)) ->      elem(V2, V4) .          origin(Derived(CCC35
      ))
DA9E7-27264 ( 1000) {} ||      elem(V2, V4)    less(V2, s(V6)) ->      elem(V2, ins(s(V6), V4)) .
      origin(Derived(CCC35))
E73B3-27265 ( 1000) {} ||      elem(V2, ins(s(V4), V6))       less(V2, s(V4)) ->      elem(V2, V6) .
      origin(Derived(CCC35))
CE01E-27266 ( 1000) {} ||      elem(V2, ins(s(V4), V6))       less(V2, s(V4)) ->      elem(V2, ins(s(V4), V6)) .
                 origin(Derived(CCC35))
DD3E9-27267 (    1) {} || ->  .                 origin(Contradicting Set(F25C3, 87558, 27412) and Set(DBE1F, DA9E7,
      CE01E, E73B3)))
```

```
8E937-27268 (    1) {} ||  ->  .                  origin(Induction(List[z by C9099, s by DD3E9]))
71276-27269 ( 1000) {} ||       elem(V2, V4)      less(V2, V6) ->         elem(V2, ins(V6, V4)) .
      origin(Derived(8E937))
3A852-27270 ( 1000) {} ||       elem(V2, ins(V4, V6))   less(V2, V4) ->         elem(V2, ins(V4, V6)) .
             origin(Derived(8E937))
904F3-27271 ( 1000) {} ||       elem(V2, V4)      less(V2, V6) ->         elem(V2, V4) .          origin(Derived(8E937
      ))
6E1FE-27272 ( 1000) {} ||       elem(V2, ins(V4, V6))   less(V2, V4) ->         elem(V2, V6) .          origin(
      Derived(8E937))
DCD34-27273 (    1) {} ||  ->  .                  origin(Contradicting Set(78F19, 46111, 917BF) and Set(3A852, 6E1FE,
      71276, 904F3)))
Involved clauses: 108

        Total time: 8.60884282 s
```

A.1.4. Problem 87

Input

The problem in DFG format, edited for better readability.

```
begin_problem(benchmark).

list_of_descriptions.
name({**}).
author({**}).
status(unknown).
description({**}).
end_of_list.


list_of_symbols.
functions [(z,0), (s,1), (minus,2)].
predicates [(less, 1+2)].
sorts [(nat,0)].
end_of_list.

list_of_declarations.
function(minus, (nat, nat) nat).
datatype(nat, z, s).
end_of_list.

list_of_formulae(axioms).
formula(forall([Y:nat, X:nat], equal:lr(minus(z,Y)      ,    z)),          minus_z_left).
formula(forall([Y:nat, X:nat], equal:lr(minus(X,z)      ,    X)),          minus_z_right).
formula(forall([Y:nat, X:nat], equal:lr(minus(s(X),s(Y)),    minus(X,Y))),minus_s).
end_of_list.


list_of_formulae(conjectures).
formula(forall([Y:nat,X:nat,Z:nat],equal(minus(minus(X,Y),Z),minus(X,minus(Y,Z)))),
      conj_87).
end_of_list.

end_problem.
```

Output

The boxed parts are the unedited output of Pirate (except for adding bold for keywords).

```
Starting
scala version 2.11.2
```

```
Unkown/Legacy−style Argument: /home/dwand/pirate/reduceDFGtoFacts
Initial Conjectures:
  : 49044−00005 (     1) {} ||    minus(minus(skf2, skf1), skf3) = minus(skf2, minus(skf1, skf3)) −>  .
        origin(Conj(conj_87 from: Set(34DDF)))
```

```
@       148  @head(0 rank: 11 ): CaseProofTask−TID:5( parent TID:4, Initial)
@       359  @head(3 rank: 26 ): CaseProofTask−TID:6( parent TID:5, Case(s of skf3))
@       630  @head(8 rank: 26 ): CaseProofTask−TID:8( parent TID:7, DeriveCase( −>   minus(minus(s(skf2), V2), skf3)
      = minus(skf2, minus(V2, s(skf3))) . ))
@       840  @head(15 rank: 24 ): CaseProofTask−TID:21( parent TID:20, Case(z of skf4))
@       903  @head(18 rank: 24 ): CaseProofTask−TID:30( parent TID:29, Case(z of skf3))
@       914  @head(17 rank: 26 ): CaseProofTask−TID:9( parent TID:7, DeriveCase( −>   minus(s(minus(skf2, skf1)), V2
     ) = minus(skf2, minus(s(skf1), V2)) . ))
@      1078  @head(22 rank: 26 ): CaseProofTask−TID:10( parent TID:7, DeriveCase( −>   minus(minus(V2, s(skf1)),
     skf3) = minus(V2, s(minus(skf1, skf3))) . ))
@      1181  @head(24 rank: 30 ): CaseProofTask−TID:31( parent TID:29, Case(s of skf3))
@      1181  @head(23 rank: 32 ): CaseProofTask−TID:24( parent TID:23, Case(z of skf2))
@      1181  @head(22 rank: 36 ): CaseProofTask−TID:33( parent TID:32, Case(z of skf2))
@      1181  @head(21 rank: 36 ): CaseProofTask−TID:39( parent TID:38, Case(z of skf5))
@      1231  @head(20 rank: 39 ): CaseProofTask−TID:12( parent TID:7, DeriveCase( −>   minus(minus(s(skf2), V2), V4)
      = minus(skf2, minus(V2, s(V4))) . ))
@      1366  @head(26 rank: 30 ): CaseProofTask−TID:56( parent TID:55, Case(z of skf8))
@      1420  @head(28 rank: 28 ): CaseProofTask−TID:65( parent TID:64, Case(z of skf7))
@      1431  @head(27 rank: 35 ): CaseProofTask−TID:66( parent TID:64, Case(s of skf7))
@      1431  @head(26 rank: 39 ): CaseProofTask−TID:14( parent TID:7, DeriveCase( −>   minus(minus(V2, s(skf1)), V4)
      = minus(V2, s(minus(skf1, V4))) . ))
@      1521  @head(29 rank: 39 ): CaseProofTask−TID:16( parent TID:7, DeriveCase( −>   minus(s(minus(skf2, skf1)),
     V2) = minus(skf2, minus(s(skf1), V2)) . ))
@      1661  @head(33 rank: 39 ): CaseProofTask−TID:26( parent TID:7, DeriveCase( −>   minus(minus(V2, skf4), skf3)
      = minus(V2, s(minus(skf4, s(skf3)))) . ))
@      1661  @head(32 rank: 39 ): CaseProofTask−TID:28( parent TID:7, DeriveCase( −>   minus(s(minus(s(skf2), skf4))
     , V2) = minus(skf2, minus(skf4, V2)) . ))
@      1662  @head(31 rank: 39 ): CaseProofTask−TID:51( parent TID:7, DeriveCase( −>   minus(s(minus(skf6, V2)), V4)
      = minus(skf6, s(minus(V2, V4))) . ))
@      1767  @head(37 rank: 44 ): CaseProofTask−TID:18( parent TID:17, Case(z of skf3))
@      1767  @head(36 rank: 44 ): CaseProofTask−TID:36( parent TID:35, Case(z of skf1))
@      1767  @head(35 rank: 45 ): CaseProofTask−TID:11( parent TID:6, Case(s of skf2))
@      2024  @head(45 rank: 20 ): CaseProofTask−TID:100( parent TID:99, Case(z of skf1))
```

```
initial found counter ex for case Case(z of skf1)

        Total time: 2.340949979 s
```

The above is the default output, used for the evaluation. Pirate can be configured to also output the counter example, the corresponding line then is:

```
initial found counter ex for case Case(z of skf1:   minus(minus(s(i1002), skf1), s(i1001)) = minus(s(i1002), minus(
     skf1, s(i1001))) −>  . )
```