TU WIEN Informatics

Constraint Superposition for Higher-order Logic

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Matthias Hetzenberger, BSc
Matrikelnummer 11775827

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof.in Dr.in techn. Laura Kovács, MSc
Mitwirkung: Dr. Alexander Bentkamp
Prof. Dr. Jasmin Blanchette

Wien, 4. September 2023

Matthias Hetzenberger | Laura Kovács

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

TU WIEN Informatics

Constraint Superposition for Higher-order Logic

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Matthias Hetzenberger, BSc
Registration Number 11775827

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Dr.in techn. Laura Kovács, MSc
Assistance: Dr. Alexander Bentkamp
Prof. Dr. Jasmin Blanchette

Vienna, 4th September, 2023

Matthias Hetzenberger	Laura Kovács

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

Erklärung zur Verfassung der Arbeit

Matthias Hetzenberger, BSc

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

Wien, 4. September 2023

Matthias Hetzenberger

Acknowledgements

First and foremost, I would like to express my gratitude towards my supervisor Laura Kovács. She was my mentor during my Bachelor with Honors program and made it possible for me to start working on the exciting topic that is presented in this thesis.

She introduced me to Alexander Bentkamp and Jasmin Blanchette who both agreed to let me work on their ongoing research together with them. I would like to thank both of them for their advice, feedback, and for taking their time to gently introduce me to their current work.

This thesis also benefited through support from people outside my academic life. I am grateful that I can always count on the help of my parents. And in the end, I would especially like to thank my wife Sonja, who always supports me.

The research grant ERC CoG 101002685 - ARTIST partially funded the work presented in this thesis.

Kurzfassung

Prädikatenlogik erster Stufe galt lange als optimale Grundlage für automatisierte Theorembeweiser in Bezug auf die Ausdrucksstärke im Verhältnis zum Automatisierungsgrad. Führende Theorembeweiser verwenden den Superpositionskalkül, welcher kürzlich auf Prädikatenlogik höherer Stufe, auch als einfache Typentheorie bezeichnet, erweitert wurde. Diese Erweiterung erlaubt es großteils, dass Beweisführung höherer Stufe auch nur für Probleme höherer Stufe verwendet wird. Die empirische Evaluierung dieses Kalküls durch den Theorembeweiser Zipperposition hat die Konkurrenzfähigkeit gegenüber anderen aktuellen Theorembeweisern für Logik höherer Stufe unter Beweis gestellt.

Obwohl der Superpositionskalkül für Logik höherer Stufe vielversprechend ist, gibt es Möglichkeiten zur Verbesserung. Durch den Sprung von Prädikatenlogik erster Stufe zu höheren Stufen wird das Unifikationsproblem unentscheidbar. Weil möglicherweise unendlich viele Unifizierer enumeriert werden müssen, kann eine Explosion des Suchraums erfolgen. G.P. Huet stellte fest, dass eine vollständige Unifikation für Kalküle höherer Ordnung nicht notwendig ist, während gleichzeitig die Eigenschaft der Widerlegungsvollständigkeit erhalten werden kann. Er führte einen Resolutionskalkül für die einfache Typentheorie ein, bei dem Constraints verwendet werden, um die Unifikation aufzuschieben. Die Unifikationsprobleme können dann mit einem Ansatz namens Preunifizierung gelöst werden, bei dem die Unifikation gestoppt werden kann, wenn es offensichtlich ist, dass eine Lösung existiert.

In dieser Arbeit wird die Idee von Huet auf den Superpositionskalkül angewandt, was den Namen *Constraint-Superpositionskalkül* ergibt. Dadurch ist es möglich, die Unifikation hinauszuzögern und die entstehenden Unifikationsconstraints erst später zu lösen. Außerdem müssen bei diesem Ansatz einige Unifizierer während des Saturationsprozesses nicht berücksichtigt werden, was zu einer Einschränkung des Suchraums führt. Aufbauend auf früheren Arbeiten präsentieren wir eine Beweisskizze für Widerlegungsvollständigkeit. Schließlich wird der in Zipperposition implementierte Ansatz erläutert und basierend auf TPTP und Sledgehammer Benchmarks evaluiert.

Abstract

For many years, first-order logic was considered the sweet spot for automated theorem provers regarding expressiveness in relation to the degree of automation. The leading theorem provers employ the superposition calculus. Recently, this calculus was successfully extended to higher-order logic, which is also called simple type theory. The extension is mostly graceful, in a sense that higher-order reasoning should exclusively be used for higher-order problems. Its empirical evaluation in the theorem prover Zipperposition proved the extension to be competitive with other current theorem provers for higher-order logic.

While the superposition calculus for higher-order logic is promising, there are possibilities for improvement. When going from first-order to higher-order logic, the unification problem becomes undecidable. Since the calculus needs to eagerly enumerate unifiers, a potential explosion in search space is possible. It was noted by G.P. Huet that full eager unification is not necessary for higher-order calculi, while still enjoying the property of refutational completeness. He introduced a resolution calculus for simple type theory, where constraints are used to postpone unification. The unification problems can then be solved using an approach called preunification, where the unification can be stopped when it is apparent that the problem admits a solution.

In this thesis, the idea of Huet is applied to the superposition calculus, which gives the name *constraint superposition calculus*. This makes it possible to postpone unification and lazily solve the arising unification constraints. Moreover, with this approach some unifiers need not be considered during saturation, which leads to a restriction of the search space. Building upon previous work, we present a proof sketch for refutational completeness. Finally, the implementation in Zipperposition is discussed and evaluated on TPTP and Sledgehammer benchmarks.

Contents

CHAPTER 1

Introduction

1.1 Motivation & Problem Statement

Nowadays, automated theorem provers are used extensively in academia and industry to verify input problems formulated within some specific logic. For example, if the logic on which a prover operates is classical propositional logic, then it is a SAT (i.e., satisfiability) solver. Applications for SAT solvers include the verification of digital circuits, scheduling problems and constraint satisfaction.

However, propositional logic is rather weak because only aspects about finite structures can be encoded and one can only use propositional variables. First-order logic is an extension of propositional logic, which solves many of these shortcomings. Using quantification, equality and function symbols, many mathematical theorems can be formalized and then be shown valid in a matter of seconds using a first-order automated theorem prover. The calculus that is used in the leading first-order provers (e.g., Vampire[1] or E[2]) is the so-called *superposition calculus* [NR01; BG94].

While many properties can be expressed in first-order logic, there are theorems that cannot be fed into a first-order prover as the needed list of axioms is infinite or quantification over predicates or functions is needed. A simple example is the arithmetic of natural numbers, which can be formalized using the well-known Peano axioms. If R is a unary predicate and n' denotes the successor of n, the induction axiom can be stated as follows:

$$R(0) \wedge \forall n.(n \in \mathbb{N} \wedge R(n) \Rightarrow R(n')) \Rightarrow \forall n.(n \in \mathbb{N} \Rightarrow R(n))$$

Note that R can be any unary predicate and thus this axiom is actually an axiom schema, i.e., for every possible unary predicate there is an instance of this axiom. Hence, a general

[1] https://vprover.github.io/
[2] http://www.eprover.org/

first-order prover cannot be given a finite axiomatization of the arithmetic of natural numbers, as it is not possible to quantify over predicates in first-order logic.

This motivates the use of *higher-order logic*. As „higher-order“ already indicates, with this logic it is possible to quantify also over predicates and functions and thus the theory of arithmetic can be finitely axiomatized and used to automatically verify many theorems. This is done by prepending a universal quantification over unary predicates R to the beginning of the above logical sentence.

To further motivate the expressiveness of higher-order logic for formalized mathematics, we consider the succinctness and automated proof of Cantor's theorem as motivated in [Ben+23a]. The theorem can be given in higher-order logic as shown below:

$$\text{surjective} = (\lambda f.\ \forall y.\ \exists x.y = f(x)) \Rightarrow \neg(\exists g : \tau \to (\tau \to \text{bool}).\ \text{surjective}(g))$$

The premise of the implication defines surjectivity of a function as a unary predicate using a λ-abstraction, which would not be possible in first-order logic. The conclusion of the implication states that there does not exist a surjective function that takes a value of some type τ and maps it to a set of values of type τ, which is represented by a function from the set of values of type τ to the Boolean type bool, which is defined to be $\{0, 1\}$. That is, the existence of a surjective function from a set to its power set can be refuted. While Cantor's set theory caused a controversy back then, today Cantor's theorem can be simply verified by an automated prover for higher-order logic [Ben+23a].

Recently, the powerful superposition calculus for first-order logic was successfully generalized to higher-order logic in a series of successive milestones. First, the calculus supported only λ-free higher-order logic [Ben+18]. Then, λ-expression were incorporated [Ben+21]. Finally, support for a Boolean type was added [Ben+23b]. The calculus was implemented in the Zipperposition[3] theorem prover and it turned out to be very efficient, as it won the CADE ATP System Competition[4] in 2020, 2021, and 2022.

There are some obstacles when moving from first-order to higher-order logic. In first-order logic, there is the notion of a unifier that rewrites variables into terms which makes two terms syntactically equal. If this is possible for two first-order terms, there is always a so-called *most-general unifier* and it is straightforward to compute one. For higher-order logic, it is not decidable whether two terms have a unifier at all and also there can be an infinite set of non-comparable unifiers. Therefore, during proof search, inferences have to be interleaved with the computation of unifiers to guarantee that a proof can eventually be found if the input problem is provable [Vuk+21].

The aim of my master thesis is to combat these issues by applying an approach that was first developed for a resolution calculus for higher-order logic by Gérard Pierre Huet in 1972 [Hue72]. In this calculus, eager higher-order unification is not needed. Instead, unification constraints are postponed as constraints and can be solved on demand. This

[3] https://github.com/sneeuwballen/zipperposition
[4] https://www.tptp.org/CASC/

brings the advantage that for certain forms of unification problems it can be easily seen that there has to be a solution without the need to enumerate every solution. Moreover, the calculus remains refutational complete with respect to so-called Henkin semantics. The goal is to exploit this idea for the higher-order superposition calculus.

1.2 Contributions

This thesis is based on unpublished[5] work of Alexander Bentkamp, Jasmin Blanchette, Uwe Waldmann, and myself. In this thesis, we present a novel superposition calculus for higher-order logic that is called *constraint superposition calculus*. Because full higher-order unification is not needed with this calculus, one can resort to the less explosive approach of preunification.

Building on previous work on the promising superposition calculus, we provide a proof sketch of refutational completeness of the calculus. In addition, we present and justify the use of various simplification rules that can be used in implementations. Moreover, the calculus was implemented in the state-of-the-art automated theorem prover Zipperposition by myself. This implementation was then evaluated against problems of the TPTP problem library as well as problems generated from Isabelle by Sledgehammer.

1.3 Structure of the Thesis

In Chapter 2 the necessary notions and concepts are introduced on which the rest of this thesis builds. After mathematical preliminaries and rewrite systems, first-order logic with interpreted Booleans, and higher-order logic using the simply typed λ-calculus are presented.

Afterwards, saturation theorem proving is discussed in Chapter 3 in which a framework is presented that allows to obtain completeness results for a calculus by instantiating the framework. In Chapter 4 a superposition calculus for first-order logic with interpreted Booleans, and a higher-order superposition calculus is discussed. The refutational completeness proofs of the calculi are presented, which both employ the saturation framework from Chapter 3. Saturation algorithms for the presented calculi are given and explained.

The constraint superposition calculus is introduced in Chapter 5. In addition to the inference rules, we provide a simple version of redundancy that allows to justify many simplification rules, which are also presented in this chapter. Moreover, a proof sketch for refutational completeness is given, and the saturation algorithm is discussed. Finally, the implementation in Zipperposition is explained and evaluated.

[5] as of August 2023

CHAPTER 2

Preliminaries

2.1 Mathematical Preliminaries

A tuple $(a_1, \ldots, a_n)$ is also written as $\bar{a}_n$ or $\bar{a}$ if the length of the tuple is irrelevant. The empty tuple is written as ε, and $\bar{a} \cdot \bar{b}$ denotes the concatentation of tuples. If $\bar{a}_n \in A^n$ and $f : A \to B$, the (slightly abusing) notation $f(\bar{a}_n)$ stands for elementwise application, i.e., $f(\bar{a}_n) = (f(a_1), \ldots, f(a_n))$. Also, if $S \subseteq A$, $f(S)$ stands for $\bigcup_{a \in S} f(a)$.

For a function $f : A \to B$, the updated function $f[a \mapsto b]$ denotes the function that behaves as f on all inputs different from a and maps a to b. That is

$$f[a \mapsto b](x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise.} \end{cases}$$

The notation $f[\bar{a}_n \mapsto \bar{b}_n]$ stands for $(\cdots((f[a_1 \mapsto b_1])[a_2 \mapsto b_2])\cdots)[a_n \mapsto b_n]$.

The composition of binary relations $R_1 \subseteq X \times Y$ and $R_2 \subseteq Y \times Z$, denoted by $R_1 \circ R_2$, is defined as follows:

$$R_1 \circ R_2 = \{(x, z) \mid \exists y.\ ((x, y) \in R_1 \text{ and } (y, z) \in R_2)\}$$

Given a binary relation R, the notation xRy may be used to denote $(x, y) \in R$.

The power of $R \subseteq S \times S$ is recursively given by $R^1 = R$ and $R^{n+1} = R \circ R^n$. Moreover, the transitive closure of $R \subseteq S \times S$, denoted by R^+, is defined by $R^+ = \bigcup_{i=1}^{\infty} R^i$, whereas the reflexive closure of R is $R \cup \{(s, s) \mid s \in S\}$. The reflexive transitive closure of R is denoted by R^* and is the union of transitive closure and reflexive closure of R.

Given a set S, the power set of S, denoted by $\mathcal{P}(S)$, is the set of all subsets of S, i.e., $\mathcal{P}(S) = \{S' \mid S' \subseteq S\}$. A *multiset* is a generalization of regular sets. Formally, a multiset M over set S is a function $M : S \to \mathbb{N}$ that maps each element a of S to the count of

occurrences in M. We write multisets like ordinary sets. A set $S' \subseteq S$ can be seen as a multiset over S with $S'(a) = 1$ if $a \in S'$ and $S'(a) = 0$ if $a \notin S'$ for all $a \in S$. In the following, let N and M be multisets over S. For example, $\{a, b, a\}$ and $\{a, a, b\}$ denote the same multiset but $\{a, b\}$ is a different multiset because the number of occurrences of a has changed. We say that a is an element of M if a occurs at least once in M, i.e., $M(a) > 0$. Like the empty set, the empty multiset over S is denoted by $\emptyset$, i.e., $\emptyset(a) = 0$ for all $a \in S$. N is included in M, written $N \subseteq M$, if $N(a) \leq M(a)$ for all $a \in S$. The union of M, N, written $M \cup N$, can simply be defined via their pointwise sum, i.e., $(M \cup N)(a) = M(a) + N(a)$ for all $a \in S$. The multiset difference of M and N, written $M - N$, is defined by $(M - N)(a) = \max\{0, M(a) - N(a)\}$.

Let $\rhd$ be a binary relation over some set S, that is, $\rhd \subseteq S \times S$. We write $a \rhd b$ instead of $(a, b) \in \rhd$ and $a \not\rhd b$ for $(a, b) \notin \rhd$. The reflexive closure of $\rhd$, denoted by $\trianglerighteq$, is defined as $\rhd \cup \{(a, a) \mid a \in S\}$. The relation $\rhd$ is *irreflexive* if $a \not\rhd a$ for all $a \in S$. It is *transitive* if $a \rhd b$ and $b \rhd c$ implies $a \rhd c$ for all $a, b, c \in S$. If either $a \rhd b$ or $b \rhd a$ holds for all $a, b \in S$, then $\rhd$ is *total*. The relation $\rhd$ is *well-founded* if there is no infinite sequence of elements $(a_i)_{i \geq 0}$ in S such that $a_0 \rhd a_1 \rhd \ldots$ holds. A *strict order* is a binary relation that is irreflexive and transitive.

Let $\rhd$ be a strict order on S. The *multiset order* over finite multisets over S induced by $\rhd$ is denoted by $\rhd_{\mathsf{mul}}$ and is defined as follows [BG01, Section 2.5]:

$$X \rhd_{\mathsf{mul}} Y \quad \text{if and only if} \quad \begin{array}{ll} (1.) & X \neq Y; \\ (2.) & \text{for all } a \in S, \text{ from } Y(a) > X(a) \text{ must follow that} \\ & \text{there is some } b \in S \text{ such that } b \rhd a \text{ and } X(b) \rhd Y(b) \end{array}$$

The order $\rhd_{\mathsf{mul}}$ is also called the *multiset extension* of $\rhd$. The multiset extension of a strict order is again a strict order. Moreover, It holds that if $\rhd$ is total or well-founded, then so is its multiset extension.

Let M be a multiset and $\trianglerighteq$ a reflexive binary relation. An element a of M is $\trianglerighteq$-*maximal* if for all $b \in M$ it holds that $b \trianglerighteq a$ implies that $a = b$. The element a is *strictly* $\trianglerighteq$-maximal if it is $\trianglerighteq$-maximal and only occurs once in M.

2.2 First-order Logic with Interpreted Booleans

This section introduces first-order logic with an interpreted Boolean type. In this logic, terms can contain Boolean connectives and quantifiers and thus a calculus for this logic does not necessarily require preprocessing steps such as clausal normal form transformation. Instead, such transformations can be performed by the calculus in a lazy way.

2.2.1 Syntax

Let Σ_{ty} be a set of types, usually denoted by τ, υ, with $o \in \Sigma_{\mathsf{ty}}$, where o is the type of Booleans. A type declaration, specifies the types of the arguments as well as the

result type. Formally, a type declaration with n arguments of type $\bar{\tau}_n$ and result type υ is represented as the tuple $(\bar{\tau}_n, \upsilon) \in \Sigma_{\mathsf{ty}}^{n+1}$. We use the more convenient notation $(\tau_1 \times \ldots \times \tau_n) \to \upsilon$, or $\bar{\tau}_n \to \upsilon$, instead of $(\bar{\tau}_n, \upsilon)$ and write υ instead of $() \to \upsilon$ for the case of $n = 0$. We say that a type declaration $\bar{\tau}_n \to \upsilon$ is *functional* if $n > 0$, and it is *nonfunctional* if $n = 0$.

Let Σ be a set of function symbols with associated type declarations. A function symbol f with type declaration $\bar{\tau}_n \to \upsilon$ is written as $\mathsf{f} : \bar{\tau}_n \to \upsilon$, or just f when the type declaration is irrelevant or can be inferred from the context. A function symbol is (non)functional if its corresponding type declaration is. We say that a nonfunctional symbol is a *constant*. A symbol with Boolean result type is a *predicate* and we also view variables of Boolean type as predicates. It is required that Σ contains at least one constant for each type $\tau \in \Sigma_{\mathsf{ty}}$. It is assumed that the logical symbols $\bot, \top : o$, $\neg : o \to o$, $\wedge, \vee, \rightarrow : (o \times o)$ are contained in Σ. The logical symbols are *interpreted*, that is, their meaning is the same in all interpretations. All other predicates, i.e., predicate symbols as well as predicate variables, are *uninterpreted* predicates, i.e., their meaning is defined by an interpretation. Additionally, for every type $\tau \in \Sigma_{\mathsf{ty}}$ the set Σ includes the logical symbols $\approx_\tau, \not\approx_\tau : (\tau \times \tau) \to o$. With slight abuse of notation, the subscript τ is omitted from $\approx$ and $\not\approx$ because it can be inferred from the arguments. The logical symbols are typeset in bold in order to emphasize that they are logical symbols and not confuse them with regular connectives $\wedge, \vee, \ldots$ of first-order logic. They are written in infix notation.

The set of variables is denoted by $\mathcal{V}$. It is assumed to be infinite and consists of variables with associated types, written as $x : \tau$ for $\tau \in \Sigma_{\mathsf{ty}}$. Note that variables can have no functional types because the logic is first-order and quantification over functions would become possible.

A pair $(\Sigma_{\mathsf{ty}}, \Sigma)$ is a *signature*. The set of first-order terms, usually denoted by t, r or s, over the signature $(\Sigma_{\mathsf{ty}}, \Sigma)$ and variables $\mathcal{V}$ is denoted by $\mathcal{T}_{\mathsf{fo}}(\Sigma_{\mathsf{ty}}, \Sigma, \mathcal{V})$. A term always has a corresponding type. We write $\bar{t}_n : \bar{\tau}_n$ if $t_i : \tau_i$ for $1 \leq i \leq n$.

The set $\mathcal{T}_{\mathsf{fo}}(\Sigma_{\mathsf{ty}}, \Sigma, \mathcal{V})$ is inductively defined as follows, making sure all terms are well-typed:

- If $x : \tau \in V$, then x is a term of type τ, i.e., $x : \tau \in \mathcal{T}_{\mathsf{fo}}(\Sigma_{\mathsf{ty}}, \Sigma, \mathcal{V})$.
- If $\mathsf{f} : \bar{\tau}_n \to \upsilon \in \Sigma$ and $t_i : \tau_i \in \mathcal{T}_{\mathsf{fo}}(\Sigma_{\mathsf{ty}}, \Sigma, \mathcal{V})$ for $1 \leq i \leq n$, then $\mathsf{f}(\bar{t}_n)$ is a term of type υ. We simply write f when $n = 0$.
- If $x : \tau \in \mathcal{V}$ and $t : o \in \mathcal{T}_{\mathsf{fo}}(\Sigma_{\mathsf{ty}}, \Sigma, \mathcal{V})$, then $\forall x.\, t$ and $\exists x.\, t$ are quantified terms and both are of Boolean type.

  Quantified terms are represented modulo α-equivalence, that is $\forall x.\, t$ and $\forall y.\, t'$, where all occurrences of x in t that are not bound by other quantifiers are replaced by y, are considered as the same quantified term.

A term of Boolean type is a *formula*. To reason about terms that occur in other terms and the locations thereof, we need to define the notions of *subterms* and *positions*. A position is represented as a finite sequence of natural numbers, where the empty tuple ε denotes the empty position. Nonempty sequences are written as $a_1.a_2.\cdots.a_n$. Moreover, let $\cdot$ denote the concatentation of sequences and let $|p| = n$ if $p = a_1.a_2.\cdots.a_n$ for $n > 0$ and $|p| = 0$ if $p = \varepsilon$.

Let t be a term. The empty position ε is a position of t and t is the subterm of t at position ε. Assume that t is the subterm of u_i at position p. Then, $i.p$ is a position of $\mathsf{f}(\bar{u})$ and t is the subterm of $\mathsf{f}\langle\bar{\tau}_n\rangle(\bar{u})$ at position $i.p$. If t is the subterm of u at position p, then $1.p$ is a position of $\mathsf{Q}x.\, u$ and t is the subterm of $\mathsf{Q}x.\, u$ at position $1.p$, where $\mathsf{Q} \in \{\forall, \exists\}$. The subterm of t at position p is denoted by $t|p$ and if the subterm u of term t at position p needs to be emphasized we write this as $t[u]_p$. We say that $t[]_p$ is a *context*, where p can be omitted if it is not important. A context $t[]_p$ has a hole at position p and filling the context with some term u is written as $t[u]_p$. Thus, $t[u]$ implies that u is a subterm of t and it may be the case that u and t are equal because every term is a subterm of itself. We say that u is a *proper subterm* of t if u is a subterm of t at some position p that is not equal to ε.

A position p is a prefix of a position q, denoted by $p \leq q$, if there exists a position t such that $q = p \cdot t$. Also, p is a *proper prefix* of q, denoted by $p < q$, if $p \leq q$ and $|p| < |q|$. If $p \leq q$, it is said that p is *at or below* q and if $p < q$ it is said that p is *below* q. For a more thorough treatment of positions and their properties we refer to [BN98]. The *root* of a term t, written $\mathtt{root}(t)$, is x if t is a variable x; f if t is of the form $\mathsf{f}(\bar{t})$ for some $\mathsf{f} \in \Sigma$; or Q if t is of the form $\mathsf{Q}x.\, t'$, where $\mathsf{Q} \in \{\forall, \exists\}$.

A *variable position* of a term t is a position p such that $t|_p$ is a variable. The *variable occurrences* of a term t is the set of variable positions of t.

A variable occurrence in a term is called *bound* if it is in the scope of some quantifier that binds the variable. More formally, if $t[x]_p$ is a variable position and there exists a position q with $q \leq p$ such that $t|_q$ is of the form $\mathsf{Q}x.\, t'$, where $\mathsf{Q} \in \{\forall, \exists\}$, we say that x occurs bound in t. Otherwise, the occurrence is called *free*. We use the standard convention that no variable occurs both bound by a quantifier as well as unbound. Moreover, we require that no variable occurs bound by different quantifiers. This can always be ensured by renaming bound variables. A term is called *ground* if it has no free variable occurrences.

A substitution is a function $\sigma : \mathcal{V} \to \mathcal{T}_{\mathsf{fo}}(\Sigma_{\mathsf{ty}}, \Sigma, \mathcal{V})$ such that the set $\mathrm{Dom}(\sigma) := \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$, called the *domain* of σ, is finite and if x is of type τ, then $\sigma(x)$ also has to be of type τ. Substitutions are usually denoted by σ, θ, ρ. It is a usual convention to write σ as $\{x_i \mapsto \sigma(x_i) \mid 1 \leq i \leq n\}$ when $\mathrm{Dom}(\sigma) = \{x_1, \ldots, x_n\}$. Moreover, $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ is written as $\{\bar{a}_n \mapsto \bar{b}_n\}$. When defining the application of a substitution σ to a term t, written as $t\sigma$, one has to ensure that substitutions are *capture avoiding*. We define the *range* of σ to be the set $\{\sigma(x) \mid x \in \mathrm{Dom}(\sigma)\}$. This means that quantified variables are not changed by substitutions. For example, if $t = (\forall x.\, \mathsf{p}(x, y)) \wedge \mathsf{q}(y)$, then the expected result of $t\{y \mapsto x\}$ is $(\forall x'.\, \mathsf{p}(x', x)) \wedge \mathsf{q}(x)$ and

not $(\forall x.\, \mathsf{p}(x, x)) \wedge \mathsf{q}(x)$.

To this end, the application $t\sigma$ is defined as follows. Let t' be the term resulting from t by replacing every subterm of the form $\mathsf{Q}x.\, u$, with $\mathsf{Q} \in \{\forall, \exists\}$, in t by $\mathsf{Q}x'.\, u'$, where x' is a new fresh variable that does not occur in t nor in the domain or the range of σ and u' is the term resulting from u in which every occurrence of x that was bound by the quantifier Q is replaced by x'. Since the only differences between t and t' are the names of the bound variables, we say that the terms are *α-equivalent*. Then, the result of the substitution $t'\sigma$ can be safely computed:

- If t' is a variable x, then $t'\sigma = \sigma(x)$.
- If t' is of the form $\mathsf{f}(t_1, \ldots, t_n)$, then $t'\sigma = \mathsf{f}(t_1\sigma, \ldots, t_n\sigma)$.
- If t' is of the form $\mathsf{Q}x.\, u$ where $\mathsf{Q} \in \{\forall, \exists\}$, then $t'\sigma = \mathsf{Q}x.\, u\sigma$. Note that this is safe, since the quantified variable x does not occur in the domain nor in the range of σ, as we use t' instead of t.

The composition of substitutions σ and θ, written $\sigma\theta$, first applies σ and then θ to the resulting term, i.e., $t(\sigma\theta) = (t\sigma)\theta$ for all terms t. If $t\sigma$ is a ground term, the substitution σ is called *grounding* for term t. A substitution σ is *more general* than a substitution θ if there exists a substitution ρ such that $\theta = \rho\sigma$. Additionally, a substitution σ is *idempotent* if and only if $\sigma = \sigma\sigma$. The identity substitution maps each variable to itself is denoted by id.

As superposition calculi are employed for logics with equality, a literal has the form of an equation $s \approx t$ or a disequation $s \not\approx t$ for terms s and t of the same type. A literal is denoted by $s \mathrel{\dot{\approx}} t$, where $\mathrel{\dot{\approx}}$ stands either for $\approx$ or $\not\approx$ and is unoriented, that is $s \mathrel{\dot{\approx}} t$ and $t \mathrel{\dot{\approx}} s$ denote the same literal. Terms t of Boolean type are not literals by themselves, and must be written as $t \mathrel{\dot{\approx}} \mathsf{T}$ or $t \mathrel{\dot{\approx}} \bot$. Such literals are called *predicate literals*. An equation literal $s \approx t$ is *positive* and a disequation literal $s \not\approx t$ is *negative*. Note that $(t \mathrel{\boldsymbol{\approx}} s) \approx \mathsf{T}$ is not the same as $(s \mathrel{\boldsymbol{\approx}} t) \approx \mathsf{T}$, even though $t \approx s$ and $s \approx t$ are equivalent. A clause is a finite multiset of literals $\{L_1, \ldots, L_n\}$ and is denoted by $L_1 \vee \ldots \vee L_n$. If $n = 0$, the clause is called empty and is written as $\bot$ and if $n = 1$ the clause is called *unit*. A clause C is subsumed by a clause D if there exists a substitution σ such that $D\sigma$ is a submultiset of C. If C is subsumed by D but D is not subsumed by C, then C is strictly subsumed by D.

The process of making two terms syntactically equal using a substitution is called *unification*. If σ is a substitution such that $t\sigma = u\sigma$ for some terms t and u, then σ is called a *unifier* of t and u. If two terms have a unifier, they are called *unifiable*. A *most general unifier* of two terms t and u, denoted by $\mathrm{mgu}(t, u)$, is a unifier σ that is more general than any other unifier of t and u. For first-order logic it is the case that there is exactly one most general unifier for two unifiable terms, and it can be computed efficiently, that is, in almost linear time w.r.t. the term size [BN98].

2.2.2 Semantics

The *universe* $\mathcal{U}$ is a mapping from each type $\tau \in \Sigma_{\mathsf{ty}}$ to its *domain* written $\mathcal{U}_\tau$. It must hold that $\mathcal{U}_\tau$ is non-empty for every $\tau \in \Sigma_{\mathsf{ty}}$ and also that $\mathcal{U}_o = \{0, 1\}$. Moreover, let $\mathcal{J}$ be an *interpretation function* that assigns every symbol $\mathsf{f} : \overline{\tau} \to \upsilon$ a function $J(f) : \mathcal{U}_{\overline{\tau}} \to \mathcal{U}_\upsilon$. That is, $\mathcal{J}(\mathsf{f})(\overline{a}) \in \mathcal{U}_\upsilon$ for all $\overline{a} \in \mathcal{U}_{\overline{\tau}}$. An interpretation $\mathcal{I}$ is a pair comprising a universe $\mathcal{U}$ and an interpretation function $\mathcal{J}$, i.e., $\mathcal{I} = (\mathcal{U}, \mathcal{J})$. It is assumed that the following requirements are satisfied for every interpretation $\mathcal{I}$, with $a, b \in \mathcal{U}_o$ and $c, d \in \mathcal{U}_\tau$ for $\tau \in \Sigma_{\mathsf{ty}}$:

(I1) $\mathcal{J}(\mathsf{T}) = 1$

(I2) $\mathcal{J}(\mathsf{\bot}) = 0$

(I3) $\mathcal{J}(\neg)(a) = 1 - a$

(I4) $\mathcal{J}(\wedge)(a, b) = \min\{a, b\}$

(I5) $\mathcal{J}(\vee)(a, b) = \max\{a, b\}$

(I6) $\mathcal{J}(\rightarrow)(a, b) = \max\{1 - a, b\}$

(I7) $\mathcal{J}(\approx_\tau)(a, b) = 1$ if $a = b$ and 0 otherwise

(I8) $\mathcal{J}(\not\approx_\tau)(a, b) = 1 - \mathcal{J}(\approx_\tau)(a, b)$

In order to deal with free variables we need a *valuation* ξ, which maps each variable $x : \tau \in \mathcal{V}$ to an element $\xi(x) \in \mathcal{U}_\tau$. Given an interpretation $\mathcal{I}$ and valuation ξ, a term $t : \tau$ can be assigned a *denotation*, written $[\![t]\!]^\xi_\mathcal{I} \in \mathcal{U}_\tau$. The denotation $[\![t]\!]^\xi_\mathcal{I}$ for $t : \tau$ is defined inductively:

- If t is a variable $x : \tau$, then $[\![x]\!]^\xi_\mathcal{I} = \xi(x)$.
- If t is of the form $\mathsf{f}(\overline{t})$ for $\mathsf{f} \in \Sigma$, then $[\![\mathsf{f}(\overline{t})]\!]^\xi_\mathcal{I} = \mathcal{J}(\mathsf{f})([\![\overline{t}]\!]^\xi_\mathcal{I})$.
- If t is of the form $\forall x.\, t'$ where $x : \tau \in \mathcal{V}$ and $t' : o$, then

$$[\![\forall x.\, t']\!]^\xi_\mathcal{I} = \min\{[\![t']\!]^{\xi[x \mapsto a]}_\mathcal{I} \mid a \in \mathcal{U}_\tau\}.$$

- If t is of the form $\exists x.\, t'$ where $x : \tau \in \mathcal{V}$ and $t' : o$, then

$$[\![\exists x.\, t']\!]^\xi_\mathcal{I} = \max\{[\![t']\!]^{\xi[x \mapsto a]}_\mathcal{I} \mid a \in \mathcal{U}_\tau\}.$$

If t is ground, the valuation cannot influence the denotation. Therefore we can write $[\![t]\!]_\mathcal{I}$ for a ground term t. A For an interpretation $\mathcal{I}$ and valuation ξ, an equation $t \approx s$ is true, if and only if $[\![t]\!]^\xi_\mathcal{I}$ and $[\![s]\!]^\xi_\mathcal{I}$ are equal. Then, a disequation $t \not\approx s$ is true if and

only if $t \approx s$ is false. A clause is true under $\mathcal{I}$ and ξ if one of the literals of the clause is true. A set of clauses is true if every clause is true. A *model* of a set of clauses N is an interpretation $\mathcal{I}$ such that N is true under $\mathcal{I}$ for every valuation ξ. If this is the case, we write $\mathcal{I} \models N$.

If every model of a set of claues M is also a model of a set of clauses N, we say that M *entails* N and write $M \models N$. A formula $t : o$ is a *tautology* if $[\![t]\!]^{\xi}_{\mathcal{I}} = 1$ for all interpretations $\mathcal{I}$ and valuations ξ.

2.3 Term Rewrite Systems

In this thesis, term rewrite systems are defined for first-order terms as defined in the previous section. Formally, a term rewrite system, often denoted by R, is a set of pairs of terms which are written as $s \longrightarrow t$. Every such element of R is called a *rewrite rule* and it must be the case that s is not a variable and $\mathcal{FV}(t) \subseteq \mathcal{FV}(s)$. A term rewrite system R induces a reduction relation $\longrightarrow_R$ between first-order terms, which is defined as follows. That is, $s \longrightarrow_R t$ if there is a rule $s' \longrightarrow t' \in R$, a position p in s, and a substition θ such that $s|_p = s'\theta$ and $t = s[t'\theta]_p$ [Ben21].

2.4 Simply Typed λ-Calculus

I introduce the simply typed λ-calculus as defined and used in [Ben+23b]. Let $\mathcal{V}_{\mathsf{ty}}$ be an infinite set of type variables, usually denoted by α, β, and Σ_{ty} be a set of type constructors with a corresponding arity which represents the number of arguments the type constructor expects. It is assumed that Σ_{ty} contains at least a binary function type constructor $\rightarrow$.

The set of types over Σ_{ty} and $\mathcal{V}_{\mathsf{ty}}$, denoted by $\mathcal{T}\mathit{ypes}(\Sigma_{\mathsf{ty}}, \mathcal{V}_{\mathsf{ty}})$, is inductively defined as follows:

- Every type variable $\alpha \in \mathcal{V}_{\mathsf{ty}}$ is a type, i.e., $\alpha \in \mathcal{T}\mathit{ypes}(\Sigma_{\mathsf{ty}}, \mathcal{V}_{\mathsf{ty}})$.
- If $\kappa \in \Sigma_{\mathsf{ty}}$ is an n-ary type constructor, and $\bar{\tau}_n$ are types, i.e., $\tau_i \in \mathcal{T}\mathit{ypes}(\Sigma_{\mathsf{ty}}, \mathcal{V}_{\mathsf{ty}})$ for $1 \leq i \leq n$, then $\kappa(\bar{\tau}_n) \in \mathcal{T}\mathit{ypes}(\Sigma_{\mathsf{ty}}, \mathcal{V}_{\mathsf{ty}})$.

If κ is a type constructor with arity 0, κ may be written instead of $\kappa()$. And as usual, the type constructor $\rightarrow$ is written using infix notation and is right-associative. That is, $\rightarrow (\alpha, \beta)$ is written as $\alpha \rightarrow \beta$ and $\alpha \rightarrow \beta \rightarrow \gamma$ denotes $\alpha \rightarrow (\beta \rightarrow \gamma)$. Moreover, $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \upsilon$ is abbreviated as $\bar{\tau}_n \rightarrow \upsilon$. Since types can be seen as first-order terms, where type variables represent variables, first-order substitutions are also defined for types. Here, a type substitution maps type variables to types.

If τ be a type and let $\bar{\alpha}_n$ be all the type variables that occur in τ in order of first occurrrence, then $\Pi\bar{\alpha}_n.\ \tau$ is a *type declaration*.

Type variables are used for polymorphism, which allows to express functions that work on all possible types instead of defining a function for every different type. Polymorphism is demonstrated by the following example.

Example 1

Consider the popular map function used extensively in functional programming, which constructs a new list by applying a mapping function to every element of the given list. To model lists, we assume that a unary type constructor list is present. Furthermore, we want to use a symbol nil to denote the empty list and cons to prepend an element to a list, as used in languages like Lisp or Scheme. In order to support lists of every possible type, we use type variables:

$$\begin{aligned} \mathsf{nil} &: \Pi\alpha.\, \mathsf{list}\langle\alpha\rangle \\ \mathsf{cons} &: \Pi\alpha.\, \alpha \to \mathsf{list}\langle\alpha\rangle \to \mathsf{list}\langle\alpha\rangle \\ \mathsf{map} &: \Pi\alpha, \beta.\, (\alpha \to \beta) \to \mathsf{list}\langle\alpha\rangle \to \mathsf{list}\langle\beta\rangle \end{aligned}$$

Using this design, the functions can be used for every possible concrete type τ.

Since in type declarations type variables can only be introduced at the beginning, this version of polymorphism is called *rank-1* polymorphism.

To define terms, let $\mathcal{V}$ be a set of term variables with corresponding types. If x is a term variable with type τ we write $x : \tau$. It is required that there is an infinite amount of variables for each type. Let Σ be a set of function symbols with a corresponding type declaration. Σ is called a *term signature* and symbols of Σ are usually denoted by $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{f}, \mathsf{g}, \mathsf{h}$. The notation $\mathsf{f} : \Pi\bar{\alpha}_n.\, \tau$ denotes that function symbol f has type declaration $\Pi\bar{\alpha}_n.\, \tau$. If $n = 0$, then $\mathsf{f} : \tau$ is written instead.

Terms are defined using three layers of abstraction: raw λ-terms, λ-terms, and terms. The bottom layer are raw λ-terms. Building on that, λ-terms will be defined as α-equivalence classes of raw λ-terms. And moreover, terms will be $\beta\eta$-equivalence classes of λ-terms.

A raw λ-term t of type τ, written as $t : \tau$, is inductively defined.

- Every variable $x : \tau \in \mathcal{V}$ is a raw λ term of type τ.
- If $\mathsf{f} : \Pi\bar{\alpha}_n.\, \tau \in \Sigma$ and $\bar{\upsilon}_n$ is a tuple of types, then $\mathsf{f}\langle\bar{\upsilon}_n\rangle$ is a raw λ-term of type $\tau\{\bar{\alpha}_n \mapsto \bar{\upsilon}_n\}$. The elements of $\bar{\upsilon}_n$ are called the *type arguments* and if f expects no type arguments, i.e., $n = 0$, then f is written instead of $\mathsf{f}\langle\rangle$.
- If $x : \tau$ is a term variable and $t : \upsilon$ is a raw λ-term, then the λ-expression $\lambda x.\, t$ is a raw λ-term of type $\tau \to \upsilon$. For nested λ-expressions, $\lambda\bar{x}_n.\, t$ is written instead of $\lambda x_1.\, \ldots \lambda x_n.\, t$.
- Let $s : \tau \to \upsilon$ and $t : \tau$ be raw λ-terms. The *application* $s\, t$ is a raw λ-term of type υ. Applications associate to the left, that is, $s\, t\, u$ means $(s\, t)\, u$.

Every raw λ-term can be written as $t\, s_1 \ldots s_n$, where $n \geq 0$, with a so-called *head* t that is not an application using the spine notation [CP03]. Given raw λ-terms s and t, s is a *subterm* of t, written as $t[s]$ if

- $t = s$; or
- $t = \lambda x.\, u[s]$ for some raw λ-term u and term variable x; or
- $t = (u[s])\, v$ for some raw λ-terms u and v; or
- $t = u\, (v[s])$ for some raw λ-terms u and v.

We say that s is a *proper* subterm of t if $s \neq t$. The set of free variables of a raw λ-term t, denoted by $\mathcal{FV}(t)$, is given as follows:

$$\begin{aligned} \mathcal{FV}(\mathsf{f}\langle\bar{\tau}_n\rangle) &= \emptyset \quad \text{for some } f \in \Sigma \\ \mathcal{FV}(x) &= \{x\} \quad \text{for some } x \in \mathcal{V} \\ \mathcal{FV}(\lambda x.\, t) &= \mathcal{FV}(t) \setminus \{x\} \\ \mathcal{FV}(t\, u) &= \mathcal{FV}(t) \cup \mathcal{FV}(u) \end{aligned}$$

If a raw λ-term t is constructed without type variables and $\mathcal{FV}(t) = \emptyset$, then t is called *ground*.

Two raw λ-terms are α-equivalent if the bound variables may be renamed in such a way that the terms become syntactically equal. This notion is captured by the α-renmaing rule, defined as $(\lambda x.\, t) \longrightarrow_\alpha (\lambda y.\, t')$, where $y \notin \mathcal{FV}(t)$ and t' results from t by replacing every occurrence of a bound variable $y : \tau$ by a fresh variable $y' : \tau$, and repacing every *free* occurrence of x in t by y. This avoids that y is captured by some λ-abstraction at a deeper level. Also only free occurrences of x must be replaced and not occurrences that are bound by another λ-abstraction that uses also x. Every raw λ-term induces an equivalence class modulo α-renaming. These classes are called λ-*terms*. A term t is called *variable-headed* if t is of the form $u\, \bar{t}_n$, where $n > 0$ and u is variable. A variable x is said to occur *applied* in a term t if there exists a subterm $x\, \bar{t}_n$, where $n > 0$.

A substitution θ is a mapping from type variables to types and from term variables to λ-terms such that only a finite amount of variables are not mapped to themselves. It is required that a substitution results in a correctly typed term and therefore it has to be the case that for each $x : \tau \in \mathcal{V}$, $x\theta$ is of type $\tau\theta$. Moreover, since λ-terms are equivalence classes of α-equivalent terms, bound variables have to be renamed before applying a substitution to avoid variable capture. That is, $(\lambda x.\, y)\{y \mapsto x\}$ must be equal to $\lambda x'.\, x$ for some fresh variable x' and not $\lambda x.\, x$. A substitution θ is called grounding w.r.t. the term t if $t\theta$ contains no free term variables and no type variables. Moreover, θ is called *monomorphizing* w.r.t. t if $t\theta$ contains no type variables. The notions of generality and idempotence are defined as for first-order terms.

The β-reduction rule is a rewrite rule defined for λ-terms as $(\lambda x.\, t) \longrightarrow_\beta t\{x \mapsto u\}$, where bound variables of t are implicitly renamed in order to avoid variable capture. A λ-term t is called *β-normal* if there exists no λ-term t' such that $t \longrightarrow_\beta t'$.

The η-reduction rewrite rule is given by $(\lambda x.\, t\, x) \longrightarrow_\eta t$, where $x \notin \mathcal{FV}(t)$. A λ-term t is called *η-short* if there exists no λ-term t' such that $t \longrightarrow_\eta t'$. A λ-term t is called *η-long* if every functional subterm is fully applied. That is, for $\mathsf{f} : \tau \to \tau \to \tau$, the term $\mathsf{f}\ \mathsf{a}$ is η-short and $\lambda x.\, \mathsf{f}\ \mathsf{a}\ x$ is η-long. Every λ-term induces an equivalence class modulo $\beta\eta$-reduction. These classes are called *terms*.

Since terms of the simply typed λ-calculus are strongly normalizing [GTL89, section 6.3.3], the reduction rules $\longrightarrow_\beta$ and $\longrightarrow_\eta$ can be exhaustively applied to a λ-term t to find a normal form, written $t{\downarrow_{\beta\eta}}$, which is used as the representative of the corresponding $\beta\eta$-equivalence class.

Sometimes it is usefull to classify the exact order of a term. To this end, the order of types is defined next, following Snyder and Gallier [SG89].

Definition 1 (Order of Types)
Let $\tau \in \mathcal{T}\mathit{ypes}(\Sigma_{\mathsf{ty}}, \mathcal{V}_{\mathsf{ty}})$ be a type. The *order* of τ, denoted by $\mathit{ord}(\tau)$ is recursively defined as

$$\mathit{ord}(\tau) = \begin{cases} \max\{\mathit{ord}(\upsilon) + 1, \mathit{ord}(\gamma)\} & \text{if } \tau = \upsilon \to \gamma \text{ for some types } \upsilon \text{ and } \gamma, \\ 1 & \text{otherwise.} \end{cases}$$

A term is of order n if the types of all occurring variables (free or bound) is at most n and the type of all occurring function symbols is at most $n+1$. Note that every functional type has an order that is at least 2. Thus, to allow functions that take only individuals as input, it is necessary to allow functions of order $n+1$ above, since functions that take individuals as arguments must be supported for first-order terms.

2.5 Higher-order Logic

Higher-order logic, also called simple type theory, is based on the simply typed lambda calculus. Given a type signature Σ_{ty} and a term signature Σ, as defined in Section 2.4, the pair $(\Sigma_{\mathsf{ty}}, \Sigma)$ is a *higher-order signature*. It is required that a nullary Boolean type constructor o is present in Σ_{ty}. Similar to Section 2.2, the following logical symbols have to be included in Σ: $\top, \bot : o$; $\neg : o \to o$; $\wedge, \vee, \to : o \to o \to o$; $\forall, \exists : \Pi\alpha.\, (\alpha \to o) \to o$; and $\approx, \not\approx : \Pi\alpha.\, \alpha \to \alpha \to o$. Additionally, the Hilbert choice operator $\varepsilon : \Pi\alpha.\, (\alpha \to o) \to \alpha$ has to be present in Σ. The binary logical symbols are usually written in infix notation, and the type argument may be omitted if is irrelevant or deducible from the context.

Given terms s and t, an equation $s \approx t$ or disequation $s \not\approx t$ is a *literal*. A literal $s \mathrel{\dot{\approx}} t$ is either an equation or a disequation. A literal of the form $s \approx t$ is called *positive*, and a literal of the form $s \not\approx t$ is called *negative*. For a literal $L = s \mathrel{\dot{\approx}} t$, define the free variables of L as $\mathcal{FV}(s) \cup \mathcal{FV}(t)$. As in first-order logic, a clause $L_1 \vee \ldots \vee L_n$ is a finite multiset of literals and the empty clause is written as $\bot$. For a clause $C = L_1 \vee \ldots \vee L_n$, define the free variables of C as $\mathcal{FV}(L_1) \cup \ldots \cup \mathcal{FV}(L_n)$.

Because interpreted quantifiers impose some problems, a nonstandard normal form, called $\beta\eta\mathsf{Q}_\eta$-normal form, is used. The $\beta\eta\mathsf{Q}_\eta$-normal form of a term t, denoted by $t{\downarrow_{\beta\eta\mathsf{Q}_\eta}}$, is defined by applying $\longrightarrow_\beta$ and $\longrightarrow_\eta$ as often as possible and finally applying the following rewrite rule Q_η exhaustively:

$$\mathsf{Q}\langle\tau\rangle\ t \longrightarrow_{\mathsf{Q}_\eta} \mathsf{Q}\langle\tau\rangle\ (\lambda x.\ t\ x)$$

where t is not a λ-expression and $\mathsf{Q} \in \{\forall, \exists\}$. When not declared otherwise, the $\beta\eta\mathsf{Q}_\eta$-normal representative is used for analyzing the structure of terms.

2.5.1 Semantics

A *universe* $\mathcal{U}$ is a collection of nonempty sets, that are also called *domains*, such that $\{0,1\} \in \mathcal{U}$. Let $\mathcal{J}_{\mathsf{ty}}$ be a mapping that assigns each n-ary type constructor κ a function $\mathcal{J}_{\mathsf{ty}}(\kappa) : \mathcal{U}^n \to \mathcal{U}$. It must be the case that $\mathcal{J}_{\mathsf{ty}}(o) = \{0,1\}$ to correctly model Booleans. Moreover, the set $\mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{D}_1, \mathcal{D}_1)$ has to be a subset of the function space from $\mathcal{D}_1$ to $\mathcal{D}_2$ for all $\mathcal{D}_1, \mathcal{D}_2 \in \mathcal{U}$. This definition is elaborated on in Section 4.2.

Together, a pair $\mathcal{I}_{\mathsf{ty}} = (\mathcal{U}, \mathcal{J}_{\mathsf{ty}})$ is called a *type interpretation*. Let ξ be a *type valuation* that assigns each type variable a domain, i.e., $\xi : \mathcal{V}_{\mathsf{ty}} \to \mathcal{U}$. The *denotation* for a type w.r.t. type interpretation $\mathcal{I}_{\mathsf{ty}}$ and type valuation ξ is defined as follows:

$$\begin{aligned}
[\![\alpha]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}} &= \xi(\alpha) \\
[\![\kappa(\bar{\tau}_n)]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}} &= \mathcal{J}_{\mathsf{ty}}(\kappa)\left([\![\tau_1]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}, \ldots, [\![\tau_n]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}\right)
\end{aligned}$$

Given a type valuation ξ, it can be extended to a *valuation* that additionally maps each term variable $x : \tau$ to an element of the denotation of its type τ, i.e., $\xi(x) \in [\![\tau]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}$. An *interpretation function* $\mathcal{J}$ takes as input a symbol $\mathsf{f} : \Pi\bar{\alpha}_n.\ \tau$ and a tuple of domains $\bar{\mathcal{D}}_n \in \mathcal{U}^n$ and assigns them a value of the denotation of the type τ using the type valuation ξ to determine the types of the type variables. That is, $\mathcal{I}(\mathsf{f}, \bar{\mathcal{D}}_n) \in [\![\tau]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}$.

As in Section 2.2, the interpretation has to obey several requirements regarding the logical symbols. To this end, let $a, b \in \{0,1\}$, $\mathcal{D} \in \mathcal{U}$, $c, d \in \mathcal{D}$, and $f \in \mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{D}, \{0,1\})$. The following constraints have to be satisfied for every interpretation function $\mathcal{J}$:

(J1) $\mathcal{J}(\mathsf{T}) = 1$
(J2) $\mathcal{J}(\mathsf{F}) = 0$
(J3) $\mathcal{J}(\neg)(a) = 1 - a$
(J4) $\mathcal{J}(\wedge)(a,b) = \min\{a,b\}$
(J5) $\mathcal{J}(\vee)(a,b) = \max\{a,b\}$
(J6) $\mathcal{J}(\to)(a,b) = \max\{1-a,b\}$
(J7) $\mathcal{J}(\approx, \mathcal{D})(c,d) = 1$ if $c = d$ and 0 otherwise
(J8) $\mathcal{J}(\not\approx, \mathcal{D})(c,d) = 1 - \mathcal{J}(\approx, \mathcal{D})(c,d)$
(J9) $\mathcal{J}(\forall, \mathcal{D})(f) = \min\{f(a) \mid a \in \mathcal{D}\}$
(J10) $\mathcal{J}(\exists, \mathcal{D})(f) = \max\{f(a) \mid a \in \mathcal{D}\}$
(J11) $f(\mathcal{J}(\varepsilon, \mathcal{D})(f)) = \max\{f(a) \mid a \in \mathcal{D}\}$

Initially, it is allowed that λ-expressions designate arbitrary elements of the domain. Once this is defined, the interpretation can be shown to be *proper* if λ-expressions are interpreted as expected. To this end, a *λ-designation function* $\mathcal{L}$ w.r.t. a type interpretation $\mathcal{I}_{\mathsf{ty}}$

is a mapping from valuations ξ and λ-expressions of type τ to elements of $[\![\tau]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}$. An *interpretation* $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}, \mathcal{L})$ comprises a type interpretation, an interpretation function, and a λ-designation function. The denotation of a term is defined recursively as follows:

$$\begin{aligned}
[\![x]\!]^{\xi}_{\mathcal{I}} &= \xi(x) \\
[\![\mathsf{f}\langle\bar{\tau}_n\rangle]\!]^{\xi}_{\mathcal{I}} &= \mathcal{J}(\mathsf{f}, [\![\tau_1]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}, \ldots, [\![\tau_n]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}) \\
[\![s\ t]\!]^{\xi}_{\mathcal{I}} &= [\![s]\!]^{\xi}_{\mathcal{I}} \left([\![t]\!]^{\xi}_{\mathcal{I}}\right) \\
[\![\lambda x.\ t]\!]^{\xi}_{\mathcal{I}} &= \mathcal{L}(\xi, \lambda x.\ t)
\end{aligned}$$

If t is ground, the denotation stays the same for all valuations ξ. Therefore, $[\![t]\!]_{\mathcal{I}}$ may be written instead of $[\![t]\!]^{\xi}_{\mathcal{I}}$ for ground terms.

If it is the case that $[\![\lambda x.\ t]\!]^{\xi}_{\mathcal{I}}(a) = [\![t]\!]^{\xi[x\mapsto a]}_{\mathcal{I}}$ for all λ-expressions $\lambda x.\ t$ and all valuations ξ, then $\mathcal{I}$ is called a *proper* interpretation. When it is possible that a type interpretation $\mathcal{I}_{\mathsf{ty}}$ and an interpretation function $\mathcal{J}$ can be made a proper interpretation using a λ-designation function, then this designation function is unique [Fit02, Proposition 2.18].

An equation $s \approx t$ is true w.r.t. $\mathcal{I}$ and ξ if $[\![s]\!]^{\xi}_{\mathcal{I}}$ and $[\![t]\!]^{\xi}_{\mathcal{I}}$ are equal, and is false otherwise. A disequation $s \not\approx t$ is true w.r.t. $\mathcal{I}$ and ξ if $s \approx t$ is false w.r.t. $\mathcal{I}$ and ξ. A clause is true if at least a literal of the clause is true, and a set of clauses is true if all the clauses in the set are true. If under the proper interpretation $\mathcal{I}$ the set of clauses N is true under all valuations ξ, then $\mathcal{I}$ is a *model* of N, denoted by $\mathcal{I} \models N$.

With the semantics defined, Example 1 can be continued by characterizing the behavior of the map function.

Example 2

The behavior of the function map can be modelled in higher-order logic using the following two clauses:

$$\begin{aligned}
&\mathsf{map}\langle\alpha,\beta\rangle\ x\ \mathsf{nil}\langle\alpha\rangle \approx \mathsf{nil}\langle\beta\rangle \\
&\mathsf{map}\langle\alpha,\beta\rangle\ x\ (\mathsf{cons}\langle\alpha\rangle\ y\ ys) \approx \mathsf{cons}\langle\beta\rangle\ (y\ x)\ (\mathsf{map}\langle\alpha,\beta\rangle\ x\ ys)
\end{aligned}$$

Note that x has type $\alpha \to \beta$, y has type α, and ys has type $\mathsf{list}\langle\alpha\rangle$. It is not necessary to add universal quantifiers because free variables can be understood to be implicitly universally quantified. Moreover, note the changes of type arguments when comparing the left-hand sides and the right-hand sides. For example, in the first clause, nil on the left-hand side represents an empty list with type argument α, and an empty list with type argument β on the right-hand side because of the type of map, to which it is applied to. The same effect can be seen for cons in the second clause.

Sometimes inference rules need to introduce special terms, called *Skolem terms*, that stand for objects whose existence is stated using existential quantification. To this end,

given a signature $(\Sigma, \Sigma_{\mathsf{ty}})$, a *Skolem-extended* signature $(\Sigma_{\mathsf{sk}}, \Sigma_{\mathsf{ty}})$ is defined such that $\Sigma \subseteq \Sigma_{\mathsf{sk}}$. Moreover, Σ_{sk} contains a symbol $\mathsf{sk}_{\Pi\bar{\alpha}.\,\forall\bar{x}.\,\exists z.\,t\,z} : \Pi\bar{\alpha}.\,\bar{\tau} \to \upsilon$ for all types υ, variables $z : \upsilon$, and terms $t : \upsilon \to o$ over the signature $(\Sigma_{\mathsf{sk}}, \Sigma_{\mathsf{ty}})$, where $\bar{\alpha}$ are the free type variables occurring in t and $\bar{x} : \bar{\tau}$ are the free term variables occurring in t, in first order of occurrence, respectively.

When Skolem-extended signatures are used, also the semantics need to be adapted because employing interpretations as defined above with Skolem-extended signatures leads to unsoundness. Consider, for example, the clause $\exists\langle\kappa\rangle\,(\lambda z.\,z \approx \mathsf{a}) \approx \mathsf{T}$. Clearly, this clause is valid w.r.t. $\models$, because the variable z can be replaced by a and hence the result must be T because of the semantics of $\exists$. Assume, that an inference produces $(\mathsf{sk}_{\exists z.\,z\approx\mathsf{a}} \approx \mathsf{a}) \approx \mathsf{T}$ from the original clause using Skolemization. But not every interpretation $\mathcal{I}$ interprets $\mathsf{sk}_{\exists z.\,z\approx\mathsf{a}}$ as $[\![a]\!]_{\mathcal{I}}$. Hence, the clause $(\mathsf{sk}_{\exists z.\,z\approx\mathsf{a}} \approx \mathsf{a}) \approx \mathsf{T}$ is not valid anymore w.r.t. $\models$ and we have an unsoundness issue.

To this end, *Skolem-aware* interpretations are defined. A proper interpretation $\mathcal{I}$ over a Skolem-extended signature is called Skolem-aware if it holds that $\mathcal{I} \models (\exists\langle\upsilon\rangle\,(\lambda z.\,t\,z)) \approx t\,(\mathsf{sk}_{\Pi\bar{\alpha}.\,\forall\bar{x}.\,\exists z.\,t\,z}\langle\bar{\alpha}\rangle\;\bar{x})$, where $\bar{\alpha}$ are the free type variables and $\bar{x}$ are the free term variables occurring in t in order of first occurrence, respectively. If $\mathcal{I}$ is a Skolem-aware interpretation and $\mathcal{I} \models N$ for a clause set N, then $\mathcal{I}$ is called a *Skolem-aware model* of N, denoted by $\mathcal{I} \models\!\!\approx N$.

2.6 Inference Systems

The main part of most calculi is an *inference system*. An inference system is a set of *inferences*. I define these notions as in [Wal+22]. To this end, let $\mathbf{F}$ be a set of formulas. The notation $\mathbf{F}$-inference denotes an inference using the set $\mathbf{F}$. Often $\mathbf{F}$ will be left implicit. Formally, an $\mathbf{F}$-inference is a tuple of formulas $(C_n, \ldots, C_1, C_0) \in \mathbf{F}^{n+1}$, where $n \geq 0$. Usually, an inference is denoted by ι. Let $\iota = (C_n, \ldots, C_1, C_0)$ for some formulas C_i. Formulas $C_n, \ldots, C_1$ are called the *premises* of ι, i.e., $\mathit{prems}(\iota) = (C_n, \ldots, C_1)$. The formula C_0 is the *conclusion* of ι, i.e., $\mathit{concl}(\iota) = C_0$. The rightmost premise C_1 is called *main premise* in case that $n \geq 1$. Therefore, let $\mathit{mprem}(\iota) = C_1$ if $n \geq 1$ and $\mathit{mprem}(\iota) = \mathit{undefined}$ if $n = 0$. The *side premises* of ι are formulas $C_n, \ldots, C_2$. Thus, $\mathit{sprems}(\iota) = (C_n, \ldots, C_n)$ if $n \geq 2$ and $\mathit{sprems}(\iota) = \varepsilon$ otherwise. If Inf is an $\mathbf{F}$-inference system and $N \subseteq \mathbf{F}$, then $\mathit{Inf}(N)$ denotes the set of inferences of Inf whose premises are included in N.

Usually, the set of possible inferences of an inference system is infinite. A remedy is to define a finite set of *inference rules* that have all the inferences of the inference system as their instances. An inference rule with hypothetical name Rule, premises $C_n, \ldots C_1$ and conclusion C_0 is written as

$$\frac{C_n \quad \ldots \quad C_1}{C_0}\ \textsc{Rule}$$

Inference rules may also define a list of *side conditions* that must be fulfilled in order that the resulting instantiation is a valid inference.

As an example, consider the resolution rule Res for first-order logic:

$$\frac{C \vee A \qquad D \vee \neg B}{(C \vee D)\sigma}\ \textsc{Res}$$

Here, the side conditions are that A and B are atomic formulas, C and D are clauses, and σ is the most general unifier of A and B. Note that A and B can be any atoms, and C and D can be any arbitrary clauses. But it must be the case that σ is the most general unifier of A and B, because otherwise the result would be nonsensical inferences. Thus, the inference rule has a possibly infinite set of instances.

An inference rule is called *sound* w.r.t. $\models$ if, for every inference ι that is an instance of the rule, it holds that $\mathcal{I} \models \mathit{prems}(\iota)$ implies $\mathcal{I} \models \mathit{concl}(\iota)$ for any interpretation $\mathcal{I}$. A weaker property of inference than soundness is *satisfiability preservation*. An inference rule preservers satisfiability if for every inference that is an instance of the rule, it is the case that if there is an interpretation in which all the premises are true, there is also a model of the conclusion. Note that the model of the conclusion can be another interpretation than the one used for the premises, unlike in the soundness property. Hence, soundness implies satisfied presented, but not vice versa.

CHAPTER 3

Saturation Theorem Proving

In the context of automated theorem proving, the strategy of *saturation* became widely successful since Robinson devised the resolution calculus [Rob65]. A saturation-based prover typically uses some inference system to deduce new consequences of a current set of formulas until all new consequences become redundant w.r.t. a specified criterion.

To be more specific, assume that one wants to automatically prove that a formula F is valid. The formula F could be some mathematical theorem or a property of hardware or software that needs to be verified. Denote with $\bot$ a contradictory formula that admits no satisfying interpretations. To show that F is valid, $\neg F$ is used as the input for a saturation-based theorem prover that tries to systematically derive $\bot$ from $\neg F$. If a contradiction is found, one can conclude that $\neg F$ is refuted and therefore obtains the validity of F. A prover is called *refutationally complete* if it always finds a contradiction when the input problem is unsatisfiable.

Recently, a framework for saturation theorem proving was introduced by Waldmann et al. [Wal+22] that allows users to obtain completeness proofs by applying the framework to the concrete calculus. The framework imposes some constraints on the used logic and the employed criterion that is used to decide when a formula is redundant w.r.t. a set of formulas. Since this framework is used by all the calculi described in this thesis, I will introduce the most important notions. In the following, let *Inf* be an inference system, let $\mathbf{F}$ be a set of formulas, let $N \subseteq \mathbf{F}$, and let $\models$ be the entailment relation of the used logic.

The entailment relation $\models$, also called consequence relation, must satisfy the following properties for every $N_1, N_2, N_3 \subseteq \mathbf{F}$:

(C1) $\bot \models N_1$; and

(C2) if $N_2 \subseteq N_1$, then $N_1 \models N_2$; and

(C3) if $N_1 \models C$ for all $C \in N_2$, then $N_1 \models N_2$; and

(C4) from $N_1 \models N_2$ and $N_2 \models N_3$ follows $N_1 \models N_3$.

A *redundancy criterion* for the inference system *Inf* is a pair $Red = (Red_{\mathrm{I}}, Red_{\mathrm{C}})$. The first component Red_{I} is mapping from sets of formulas to sets of inferences, i.e., $Red_{\mathrm{I}} : \mathcal{P}(\mathbf{F}) : \mathcal{P}(Inf)$. Inferences in the set $Red_{\mathrm{I}}(N)$ are said to be redundant w.r.t. N and thus need not be carried out. The second component Red_{C} is a mapping from sets of formulas to sets of formulas, i.e., $Red_{\mathrm{C}} : \mathcal{P}(\mathbf{F}) \to \mathcal{P}(\mathbf{F})$. Formulas in the set $Red_{\mathrm{C}}(N)$ are also said to be redundant w.r.t. N and can therefore be deleted. A redundancy criterion has to fulfill some constraints, that are specified next. For all sets of formulas N and N' with $N \cup N' \subseteq \mathbf{F}$ it must hold that

(R1) if $N \models \bot$, then $N \setminus Red_{\mathrm{C}}(N) \models \bot$; and

(R2) if $N \subseteq N'$, then $Red_{\mathrm{C}}(N) \subseteq Red_{\mathrm{C}}(N')$ and $Red_{\mathrm{I}}(N) \subseteq Red_{\mathrm{I}}(N')$; and

(R3) if $N' \subseteq Red_{\mathrm{C}}(N)$, then $Red_{\mathrm{C}}(N) \subseteq Red_{\mathrm{C}}(N \setminus N')$ and $Red_{\mathrm{I}}(N) \subseteq Red_{\mathrm{I}}(N \setminus N')$; and

(R4) if $\iota \in Inf$ and $concl(\iota) \in N$, then $\iota \in Red_{\mathrm{I}}(N)$.

Condition (R1) specifies that if a set of formulas is unsatisfiable, then it has also to remain unsatisfiable when the redundant formulas are removed. Formulas or inferences must stay redundant if other formulas or inferences are added, as stated by condition (R2). Also, condition (R3) says that formulas and inferences stay redundant if redundant formulas or inferences are deleted. Finally, condition (R4) asserts that an inference becomes redundant when it has been carried out.

Having defined the notion of redundancy, it is now possible to actually specify when a set of formulas is saturated. The set $N \subseteq \mathbf{F}$ is said to be *saturated* w.r.t. *Inf* and $(Red_{\mathrm{I}}, Red_{\mathrm{C}})$ if $Inf(N) \subseteq Red_{\mathrm{I}}(N)$. That is, N is saturated if all inferences of *Inf* whose premises are in N are redundant w.r.t. Red_{I}.

In the implementation of saturation-based automatic theorem provers, the redundancy criterion is not directly applied, but rather to justify *simplification rules*. Simplification rules are used to simplify formulas in the current set of formulas and to remove formulas that got redundant by carrying out the simplification rule. A simplification rule is written as

$$\frac{C_1 \quad \cdots \quad C_n}{D_1 \quad \cdots \quad D_m}$$

where $C_1, \ldots, C_n$ are the premises and $D_1, \ldots, D_m$ are the conclusions.

It is a proof obligation to show that $\{C_1, \ldots, C_n\} \subseteq Red_{\mathrm{C}}(\{D_1, \ldots, D_m\})$ in order to employ the simplification rule in a prover. Then, if the simplification rule is used in an implementation, the premises can be deleted once the corresponding conclusions are

added to the current set of formulas. In the following, I will refer to a pair (Inf, Red) consisting of an inference system and a redundancy criterion thereof simply as a calculus.

After designing the inference system and the corresponding redundancy criterion, it is a desirable goal to prove the refutational completeness of the calculus. There exist two variants of refutational completeness: static and dynamic. Static completeness refers only to sets of clauses that are saturated, while dynamic completeness uses the notion of *fair derivations*, as defined below.

Definition 2 (Statical Completeness)

The inference system *Inf* is *statically refutationally complete* w.r.t. $(Red_{\mathrm{I}}, Red_{\mathrm{C}})$ and $\models$, if for every saturated set of formulas $N \subseteq \mathbf{F}$ such that $N \models \bot$, it follows that $\bot \in N$.

Definition 3 (Dynamical Completeness)

A sequence of formulas $(N_i)_i$ is a *derivation* if $N_i \setminus N_{i+1} \subseteq Red_{\mathrm{C}}(N_{i+1})$ for every i. The *limit* of $(N_i)_i$ is $N_\infty = \bigcup_i \bigcap_{j \geq i} N_j$. A formula that is a member of the limit N_∞ is called *persistent*, because it appears in some N_i and is never deleted in any subsequent N_j with $j > i$. A derivation $(N_i)_i$ is *fair* if the set of *Inf*-inferences from clauses in N_∞ is a subset of $\bigcup_i Red_{\mathrm{I}}(N_i)$. The inference system *Inf* is *dynamically refutationally complete* w.r.t. $(Red_{\mathrm{I}}, Red_{\mathrm{C}})$ and $\models$, if for all fair derivations $(N_i)_i$ it holds that $N_0 \models \bot$ implies that there exists an i with $\bot \in N_i$.

It can be shown that static completeness implies dynamic completeness and vice versa [Wal+22, lemma 10 and 11]. Often it is easier to show the static completeness of a version of a calculus that operates only on ground formulas. This result can then be transferred to the nonground case using the framework as follows. Let $\mathbf{G}$ denote the set of ground formulas. A grounding function $\mathcal{G} : \mathbf{F} \to \mathcal{P}(\mathbf{G})$ maps a formula to its set of ground instances. Let *FInf* be an inference system that operates on $\mathbf{F}$ and let *GInf* be a $\mathbf{G}$-inference system. Moreover, let $Red = (Red_{\mathrm{I}}, Red_{\mathrm{C}})$ be a redundancy criterion for *GInf*. The grounding function $\mathcal{G}$ is extended to inferences of *FInf* by mapping an $\mathbf{F}$-inference of *FInf* either to *undef* or $\mathbf{G}$-inference in *GInf*. A grounding function must obey the following conditions:

(G1) $\mathcal{G}(\bot) = \{\bot\}$;

(G2) if $\bot \in \mathcal{G}(C)$, then $C = \bot$ for every $C \in \mathbf{F}$;

(G3) for every $\iota \in FInf$, if $\mathcal{G}(\iota) \neq undef$, then $\mathcal{G}(\iota) \subseteq Red_{\mathrm{I}}(\mathcal{G}(concl(\iota)))$.

Condition (G3) ensures that all the ground instances of the conclusion of ι make the set of ground inferences obtained by $\mathcal{G}(\iota)$ redundant. In order to obtain a redundancy criterion for *FInf* using *Red* of *GInf*, the so-called *$\mathcal{G}$-lifting* of *Red* can be used.

Given $Red = (Red_{\mathrm{I}}, Red_{\mathrm{C}})$, the $\mathcal{G}$-lifting $Red^{\mathcal{G}} = (Red^{\mathcal{G}}_{\mathrm{I}}, Red^{\mathcal{G}}_{\mathrm{C}})$ is defined such that

- $\iota \in Red^{\mathcal{G}}_{\mathrm{I}}$ if and only if
  - $\mathcal{G}(\iota) \neq \mathit{undef}$ and $\mathcal{G}(\iota) \subseteq Red_{\mathrm{I}}(\mathcal{G}(N))$; or
  - $\mathcal{G}(\iota) = \mathit{undef}$ and $\mathcal{G}(\mathit{concl}(\iota)) \subseteq \mathcal{G}(N) \cup Red_{\mathrm{C}}(\mathcal{G}(N))$.
- $F \in Red^{\mathcal{G}}_{\mathrm{C}}(N)$ if and only if $\mathcal{G}(F) \subseteq Red_{\mathrm{C}}(\mathcal{G}(N))$.

It can then be shown that $Red^{\mathcal{G}}$ satisfies conditions (R1) to (R4) of redundancy criterions if Red does.

Using a grounding function $\mathcal{G}$ the relation $\models_{\mathcal{G}}$ is defined as $N_1 \models_{\mathcal{G}} N_2$ if and only if $\mathcal{G}(N_1) \models \mathcal{G}(N_2)$. That is, $\models_{\mathcal{G}}$ is Herbrand entailment. If $\models$ fulfills all requirements of a consequence relation, so does $\models_{\mathcal{G}}$. If a result requires that $\models_{\mathcal{G}}$ is replaced by Tarski entailment, i.e., $N_1 \models N_2$ if and only if every model of N_1 is also a model of N_2, it needs to be shown that $N \models_{\mathcal{G}} \bot$ if and only if $N \models \bot$ for the concrete logic used.

When using the saturation framework, the main proof obligation for dynamic completeness of the nonground calculus is to show that every $GInf$-inference is either *liftable* to an $FInf$-inference or redundant. A $GInf$-inference whose premises are in $\mathcal{G}(N)$ is liftable if it is contained in $\mathcal{G}(FInf(N))$.

The following theorem can be used to lift static refutational completeness from a ground inference system $GInf$ and redundancy criterion Red to $FInf$ and Red^{G}.

Theorem 1 ([Wal+22])

If $(GInf, Red)$ is statically refutationally complete and $GInf(\mathcal{G}(N)) \subseteq \mathcal{G}(FInf(N)) \cup Red_{\mathrm{I}}(\mathcal{G}(N))$ for all $N \subseteq \mathbf{F}$ that are saturated w.r.t. $FInf$ and $Red^{\mathcal{G}}$, then $(FInf, Red^{\mathcal{G}})$ is statically refutationally complete w.r.t. $\models_{\mathcal{G}}$.

When proving the refutational completeness of the higher-order superposition calculus, a more involved version of Theorem 1 is used. The concrete version is stated in the respective section, with the specific inference system and redundancy criterion used.

CHAPTER 4

Superposition Calculus

Superposition [BG94] is an effective calculus for first-order logic with equality. The leading automated theorem provers, for example Vampire or E, implement variations of the superposition calculus. Building on Knuth-Bendix completion and the resolution calculus, a hypothesis F is proved from a set of axioms $\{A_1, \ldots, A_n\}$ as follows. In order to prove the formula $(A_1 \wedge \cdots \wedge A_n) \to F$ valid, one can also prove that $\neg((A_1 \wedge \cdots \wedge A_n) \to F)$ is unsatisfiable. Pushing the negation inwards, one gets $A_1 \wedge \cdots \wedge A_n \wedge \neg F$. Since the calculus operates on clauses, the formula $A_1 \wedge \cdots \wedge A_n \wedge \neg F$ is translated to clausal normal form, which then serves as the input problem. From this given set of clauses, the superposition calculus derives new clauses using a set of inference rules. If the hypothesis F follows from the axioms, then, with enough computational resources, the empty clause can be inferred. This proves that $A_1 \wedge \cdots \wedge A_n \wedge \neg F$ is unsatisfiable, and hence $(A_1 \wedge \cdots \wedge A_n) \to F$ is valid.

Section 4.1 introduces a variant of the superposition calculus for first-order logic, which adds support for an interpreted Boolean type. In Section 4.2, this calculus is extended to higher-order logic. For both calculi, the respective proof of refutational completeness is explained.

4.1 The First-order Case

In the standard superposition calculus [BG94], the input problem has to be transformed into clausal normal form, which is called *preprocessing clausification*. The superposition calculus [Num+21] presented in this section, allows that formulas can appear in clauses using an interpreted Boolean type. Moreover, the calculus allows *inprocessing clausification*, which means that rules in the calculus itself perform the clausification as needed. This has the positive effect, that the effective simplification machinery of superposition is able to operate on whole formulas. For example, subterms in the formula $s \leftrightarrow t$ can be rewritten, before the formula is transformed into $s \to t$ and $t \to s$.

The proof of refutational completeness of the calculus, discussed in Section 4.1.5, is later on used to obtain refutational completeness of the higher-order superposition calculus.

4.1.1 Term Orders

Superposition calculi employ *term orders* that are used to compare terms in order to break symmetries in the search space and therefore restrict the number of possible inferences while preserving the completeness of the calculus.

Definition 4 (Term order)

A *term order* $\succ$ is a binary relation over terms. If $t \succ s$ for some term order $\succ$, the intuition is that the term s should be in some sense simpler than t.

To this end, a term order needs to satisfy several conditions:

- The relation $\succ$ is a strict order.
- It is total and well-founded when restricted to ground terms.
- For all terms $t \in \mathcal{T}_{\mathsf{fo}}(\Sigma_{\mathsf{ty}}, \Sigma, \mathcal{V}) \setminus \{\mathsf{T}, \bot\}$ it holds that $t \succ \bot \succ \mathsf{T}$.
- Let u be a term whose only Boolean subterms are T and $\bot$. Then, $\mathsf{Q}x.t \succ t\{x \mapsto u\}$ for all terms t and with $\mathsf{Q} \in \{\mathsf{T}, \bot\}$.
- The order $\succ$ enjoys the *subterm property*. That is, for all terms t and u it holds that $t[u] \succeq u$.
- The order $\succ$ is compatible with contexts, i.e., $t \succ t'$ implies $u[t] \succ u[t']$ for all terms t, t' and u. The compatibility with contexts does not need to hold below quantifiers.
- Stability under substitutions, i.e., $s \succeq t$ implies $s\sigma \succeq t\sigma$ for all terms s, t and substitutions σ.

Concrete term orders that are often used in automated theorem provers are the *lexicographic path order* (LPO) [BN98, Definition 5.4.12] and the *Knuth-Bendix order* (KBO) [BN98, Definition 5.4.18]. KBO uses a weight function that maps a weight to symbols of Σ, as well as a precedence on Σ to compare terms. Originally, nonnegative integers are used for the weight function, but there are also variants that use ordinals [LW07; KMV11]. In contrast to KBO, LPO relies entirely on a precedence on Σ to compare terms.

Using an encoding of terms into untyped first-order logic, both LPO and transfinite KBO are able to fulfill the presented conditions. The encoding represents quantifier-bound variables by *De Bruijn indices* (see Section 5.3), which are implemented by fresh constant symbols db_n, where $n \in \mathbb{N}$ is the value of the index. Function symbols and logical symbols are just mapped to their untyped counterparts. Quantifiers are mapped to fresh unary

function symbols $\mathsf{Q}_\forall$ and $\mathsf{Q}_\exists$. The encoding can then be defined using the following function $\mathtt{enc}$ that computes the resulting recursively depending on the structure of the given term:

$$\begin{aligned}
\mathtt{enc}(t, \sigma, \ell) &= t\sigma && \text{if } t \text{ is a variable} \\
\mathtt{enc}(\mathsf{f}\langle\bar{\tau}_n\rangle(\bar{t}), \sigma, \ell) &= \mathsf{f}\langle\bar{\tau}_n\rangle(\mathtt{enc}(\bar{t}, \sigma, \ell)) && \text{if } f \text{ is a function symbol or a logical symbol} \\
\mathtt{enc}(\forall x.\, t, \sigma, \ell) &= \mathsf{Q}_\forall(\mathtt{enc}(t, \sigma[x \mapsto \mathsf{db}\langle\bar{\tau}_n\rangle_\ell], \ell + 1)) \\
\mathtt{enc}(\exists x.\, t, \sigma, \ell) &= \mathsf{Q}_\exists(\mathtt{enc}(t, \sigma[x \mapsto \mathsf{db}\langle\bar{\tau}_n\rangle_\ell], \ell + 1))
\end{aligned}$$

The function $\mathtt{enc}$ maintains a substitution σ that maps quantified variables to their De Bruijn index and a level ℓ that must be used for the next deeper quantifiers. When encoding a term t, the result of $\mathtt{enc}(t, \mathsf{id}\langle\bar{\tau}_n\rangle, 0)$ needs to be computed. If the recursion encounters a variable x the result of $x\sigma$ is either a De Bruijn index because x is bound by a quantifier, or it is not replaced by σ because it is a free variable of the given term. Note that when updating the current substitution, no previous binding can be lost because it is assumed that every quantifier binds a unique variable.

4.1.2 Selection Functions and Eligibility

Two selection functions are needed to parameterize the calculus.

Definition 5 (Selection Functions)
A literal selection function $LSel$ is a mapping from clauses to a subset of the literals of the clause and a Boolean subterm selection function $BSel$ is a mapping from clauses to a subset of positions of Boolean subterms in the clause. Literals $LSel(C)$ and positions $BSel(C)$ are called *selected* in C. Both functions must obey the given restrictions:

- Only negative literals can be selected.
- A Boolean subterm may only be selected if it is not included in $\{\mathsf{T}, \bot\}$ and is not a variable.
- A selected Boolean subterm does not occur below a quantifier.

The published version of the presented calculus contains a mistake that allows the selection of positive literals if they are of the form $s \equiv \bot$ [Num+21] and was discovered by Yicheng Qian. But the completeness theorem does not hold if this is allowed and thus the selection of literals of the form $s \equiv \bot$ must not be possible, see also the errata of Alexander Bentkamp's PhD thesis [Ben23].

The notion of *eligibility* defines when inferences are possible using the literal and the Boolean subterm selection function as well as the term order.

Definition 6 (Eligibility)

A literal L is *(strictly) $\succeq$-eligible* w.r.t. a substitution σ in a clause C if it is selected in C or there are no selected literals and no selected Boolean subterms in C and $L\sigma$ is (strictly) $\succeq$-maximal in $C\sigma$. The definition of the set of $\succeq$-*eligible positions* of a clause C w.r.t. a substitution σ is given below:

- All selected positions are $\succeq$-eligible.
- If $s \mathrel{\dot{\approx}} t$ with $s\sigma \not\preceq t\sigma$ is either $\succeq$-eligible and negative or strictly $\succeq$-eligible and positive, then $L.s.\varepsilon$ is $\succeq$-eligible.
- If position p is eligible w.r.t. C and σ and the root of $(C|_p)\sigma$ is not included in $\{\approx, \not\approx, \forall, \exists\}$, then the positions of all direct subterms are also eligible.
- If position p is eligible w.r.t. C and σ and $(C|_p)\sigma$ is of the form $s \approx t$ or $s \not\approx t$, the position of s is eligible if $s\sigma \not\preceq t\sigma$ and the position of t is eligible if $s\sigma \not\succeq t\sigma$.

4.1.3 Inference Rules

In the following, the inference rules of the calculus are defined. The set of inference rules is subsequently denoted by $FInf$. A single rule is given in a box, where the top part shows the inference rule and the bottom part specifies the corresponding side conditions.

$$\frac{\overbrace{D' \vee t \approx t'}^{D} \qquad C[u]}{(D' \vee C[t'])\sigma}\ \textsc{Sup}$$

1. $\sigma = \mathrm{mgu}(t, u)$;
2. u is not a variable;
3. $t\sigma \not\preceq t\sigma'$;
4. $D \prec C$;
5. the position of u is $\succeq$-eligible in C w.r.t. σ;
6. $t \approx t'$ is strictly $\succeq$-eligible in D with respect to σ;
7. the root of t is not a logical symbol;
8. if $t'\sigma = \bot$, the subterm u is at the top lovel of a positive literal.

$$\frac{\overbrace{C' \vee u' \approx v' \vee u \approx v}^{C}}{(C' \vee v \not\approx v' \vee u \approx v')\sigma}\ \textsc{Factor}$$

1. $\sigma = \mathrm{mgu}(u, u')$;
2. $u\sigma \not\approx t \notin C\sigma$ for any term t;
3. no Boolean subterm and no literal is selected in C;
4. $u\sigma$ is a $\succeq$-maximal term in $C\sigma$;
5. $v\sigma$ is $\succeq$-maximal in $\{t \mid u\sigma \approx t \in C\sigma\}$.

$$\frac{\overbrace{C' \vee u \not\approx u'}^{C}}{C'\sigma}\ \textsc{Irrefl}$$

1. $\sigma = \mathrm{mgu}(u, u')$;
2. $u \not\approx u'$ is $\succeq$-eligible in C with respect to σ.

$$\frac{\overbrace{C' \vee u \not\approx u'}^{C}}{C'\sigma}\ \bot\textsc{Elim}$$

1. $\sigma = \mathrm{mgu}(s \approx t, \bot \approx \top)$;
2. $s \approx t$ is strictly $\succeq$-eligible in C with respect to σ.

$$\frac{C[u]}{C[t']\sigma}\ \textsc{BoolRw}$$

1. (t, t') is one of the following pairs, where x is a fresh variable:

$(\neg\bot, \top)$	$(\bot \wedge \top, \bot)$	$(\bot \vee \top, \top)$	$(\bot \to \top, \top)$
$(\neg\top, \bot)$	$(\top \wedge \top, \top)$	$(\top \vee \top, \top)$	$(\top \to \top, \top)$
$(\bot \wedge \bot, \bot)$	$(\bot \vee \bot, \bot)$	$(\bot \to \bot, \top)$	$(x \approx x, \top)$
$(\top \wedge \bot, \bot)$	$(\top \vee \bot, \top)$	$(\top \to \bot, \bot)$	$(x \not\approx x, \bot)$

2. $\sigma = \mathrm{mgu}(t, u)$;
3. u is not a variable;
4. the position of u is $\succeq$-eligible in C with respect to σ.

$$\frac{C[\forall z.\, v]}{C[v\{z \mapsto \mathsf{sk}\langle \bar{\tau}_n \rangle_{\forall \bar{y}.\exists z.\neg v}(\bar{y})\}]}\ \forall\text{Rw}$$

1. $\bar{y}$ are the free variables occurring in $\forall z.\, v$ in order of first appearance;
2. the position of $\forall z.\, v$ is $\succeq$-eligible in C;
3. $C[\mathsf{T}]$ is not a tautology.

$$\frac{C[\exists z.\, v]}{C[v\{z \mapsto \mathsf{sk}\langle \bar{\tau}_n \rangle_{\forall \bar{y}.\exists z.v}(\bar{y})\}]}\ \exists\text{Rw}$$

1. $\bar{y}$ are the free variables occurring in $\exists z.\, v$ in order of first appearance;
2. the position of $\exists z.\, v$ is $\succeq$-eligible in C;
3. $C[\bot]$ is not a tautology.

$$\frac{C[u]}{C[\bot] \vee u \approx \mathsf{T}}\ \textsc{BoolHoist}$$

1. u is of Boolean type and $\mathtt{root}(u)$ is an uninterpreted predicate;
2. the position of u is $\succeq$-eligible in C;
3. u is not a variable;
4. u is not at the top level of a positive literal.

$$\frac{C[s \mathrel{\approx\!\!\!\approx} t]}{C[\bot] \vee s \approx t}\ \mathrel{\approx\!\!\!\approx}\textsc{Hoist}$$

1. the position of $s \mathrel{\approx\!\!\!\approx} t$ is $\succeq$-eligible in C.

$$\frac{C[s \not\approx\!\!\!\approx t]}{C[\mathsf{T}] \vee s \approx t}\ \not\approx\!\!\!\approx\textsc{Hoist}$$

1. the position of $s \not\approx\!\!\!\approx t$ is $\succeq$-eligible in C.

$$\frac{C[\forall x.\, t]}{C[\bot] \vee t\{x \mapsto y\} \approx \mathsf{T}}\ \forall\textsc{Hoist}$$

1. the position of $\forall x.\, t$ is $\succeq$-eligible in C;
2. y is a fresh variable.

$$\frac{C[\exists x.\, t]}{C[\mathsf{T}] \vee t\{x \mapsto y\} \approx \bot}\ \exists\textsc{Hoist}$$

1. the position of $\exists x.\, t$ is $\succeq$-eligible in C;
2. y is a fresh variable.

Most of the inferences of *FInf* are sound, but not all of them. Since the rules ∀Rw and ∃Rw introduce Skolem symbols, both of these rules are only satisfiability preserving under the assumption that no Skolem symbols occur in the initial formula.

Example 3 (Drinker Paradox)

Using the *Drinker Paradox*, I will illustrate how to use some of these inference rules in practice [Smu11]. The drinker paradox is a theorem in first-order logic whose natural language representation says the following: „In a bar there is someone such that if he or she is drinking, it follows that everyone in the bar is drinking“. In first-order logic the statement is defined as $\exists x.\, (\mathsf{drinks}(x) \to \forall y.\, \mathsf{drinks}(y))$, which is a term of Boolean type in the presented logic. This statement seems paradoxical because of the use of natural language if-then statements for logical implications, which causes one to think that there is a person who is the cause for the intoxication of every person in the bar. To show that the formula is valid, the empty clause needs to be derived from $\exists x.\, (\mathsf{drinks}(x) \to \forall y.\, \mathsf{drinks}(y)) \approx \bot$.

In the derivations for this example the selection functions and term order were left implicit. First, from $\exists x.\, (\mathsf{drinks}(x) \to \forall y.\, \mathsf{drinks}(y)) \approx \bot$ the clause $\mathsf{drinks}(z) \approx \mathsf{T}$ can be obtained by the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\exists x.\, (\mathsf{drinks}(x) \to \forall y.\, \mathsf{drinks}(y)) \approx \bot}{\mathsf{T} \approx \bot \vee \mathsf{drinks}(z) \to \forall y.\, \mathsf{drinks}(y) \approx \bot}\,\exists\textsc{Hoist}}{\mathsf{drinks}(z) \to \forall y.\, \mathsf{drinks}(y) \approx \bot}\,\bot\textsc{Elim}}{\mathsf{drinks}(z) \to \bot \approx \bot \vee \mathsf{drinks}(y') \approx \mathsf{T}}\,\forall\textsc{Hoist}}{\bot \to \bot \approx \bot \vee \mathsf{drinks}(y') \approx \mathsf{T} \vee \mathsf{drinks}(z) \approx \mathsf{T}}\,\textsc{BoolHoist}}{\mathsf{T} \approx \bot \vee \mathsf{drinks}(y') \approx \mathsf{T} \vee \mathsf{drinks}(z) \approx \mathsf{T}}\,\textsc{BoolRw}}{\mathsf{drinks}(y') \approx \mathsf{T} \vee \mathsf{drinks}(z) \approx \mathsf{T}}\,\bot\textsc{Elim}}{\mathsf{drinks}(z) \approx \mathsf{T}}\,\textsc{Irrefl}$$

The unit clause $\mathsf{drinks}(z) \approx \mathsf{T}$ can then be used to superpose into the clause $\mathsf{drinks}(z) \to \forall y.\, \mathsf{drinks}(y) \approx \bot$ resulting in $\mathsf{T} \to \forall y.\, \mathsf{drinks}(y) \approx \bot$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{drinks}(z) \approx \mathsf{T} \qquad \dfrac{\mathsf{drinks}(x) \approx \mathsf{T} \qquad \mathsf{drinks}(z) \to \forall y.\, \mathsf{drinks}(y) \approx \bot}{\mathsf{T} \to \forall y.\, \mathsf{drinks}(y) \approx \bot}\,\textsc{Sup}^*}{\mathsf{T} \to \mathsf{drinks}(\mathsf{sk}\langle \bar{\tau}_n \rangle_{\exists y.\, \neg \mathsf{drinks}(y)}) \approx \bot}\,\forall\textsc{Rw}}{\mathsf{T} \to \mathsf{T} \approx \bot}\,\textsc{Sup}}{\mathsf{T} \approx \bot}\,\textsc{BoolRw}}{\bot}\,\bot\textsc{Elim}$$

In the Sup inference that is annotated with *, the unifier $\{x \mapsto z\}$ is used, because the clause $\mathsf{drinks}(z) \approx \mathsf{T}$ was changed into $\mathsf{drinks}(x) \approx \mathsf{T}$ to ensure that all premises of the inference are variable disjoint. After the first Sup inference, the subterm $\forall y.\, \mathsf{drinks}(y)$ needs to be rewritten using $\forall$Rw in order to be able to unify with $\mathsf{drinks}(z)$ such that the second Sup inference is possible. Otherwise, the terms would not be unifiable because of the leading quantifier. The application of $\forall$Rw was justified because $\mathsf{T} \to \mathsf{T} \approx \bot$ is clearly not a tautology.

4.1.4 The Redundancy Criterion

In superposition calculi, the *redundancy criterion* is a powerful tool that allows to remove redundant clauses from a given clause set. As in [BG01] the redundancy criterion is first developed for ground clauses and inferences and then lifted to their nonground counterparts. This separation is also needed for the refutational completeness proof. To this end, a ground inference system *GInf* is introduced, whose inferences operate on ground clauses. Paramters of *GInf* are a literal selection function *GLSel*, a Boolean subterm selection function *GBSel*, and a witness function $\mathbf{w}$. The same restrictions are in place for *GLSel* and *GBSel* as in Definition 5. The witness function $\mathbf{w}$ takes a ground clause C and a subterm $\exists x.\, v$ as input and produces a ground term $\mathbf{w}(C, \exists x.\, v)$. In the inference rules concerned with quantifiers, this function will provide Skolem terms. Let Q be the set of all triples $(\mathit{GLSel}, \mathit{GBSel}, \mathbf{w})$ that fulfill the restrictions and let $q \in Q$ be a *parameter triple*. We write GInf^q to specify that the inference system *GInf* is parameterized with the parameter triple q. The term order is also a parameter of *GInf*, but the order stays the same on the ground level and is therefore not explicitly given.

GInf^q consists of the inference rules Sup, Factor, Irrefl, $\bot$Elim, BoolRw, $\approx$Hoist, $\not\approx$Hoist, GforallRw, GexistsRw, G$\forall$Hoist, and G$\exists$Hoist on ground clauses. The rules GforallRw, GexistsRw, G$\forall$Hoist, and G$\exists$Hoist are specific to the ground calculus and specified below.

$$\frac{C[\forall x.\, v]}{C[v\{x \mapsto \mathbf{w}(C, \exists x.\, \neg v)\}]}\ \textsc{GforallRw}$$

$$\frac{C[\exists x.\, v]}{C[v\{x \mapsto \mathbf{w}(C, \exists x.\, v)\}]}\ \textsc{GexistsRw}$$

1. the position of the indicated subterm of C is $\succeq$-eligible in C;
2. $C[t]$ is not a tautology, where
   - $t = \top$ for GforallRw;
   - $t = \bot$ for GexistsRw.

$$\frac{C[\forall x.\, v]}{C[\bot] \vee t\{x \mapsto u\} \approx \top}\ \textsc{G}\forall\textsc{Hoist}$$

$$\frac{C[\exists x.\, v]}{C[\top] \vee t\{x \mapsto u\} \approx \bot}\ \textsc{G}\exists\textsc{Hoist}$$

1. u is a ground term whose Boolean subterms are a subset of $\{\top, \bot\}$;
2. the position of the indicated subterm of C is $\succeq$-eligible in C.

Let C be a ground clause and N be a ground clause set. The set of redundant clauses

induced by N is denoted by $GRed_{\mathrm{C}}(N)$. We define that $C \in GRed_{\mathrm{C}}(N)$ if $\{D \in N \mid D \prec C\} \models C$. The set of redundant inferences with respect to N is denoted by $GRed_{\mathrm{I}}(N)$. Let ι be an inference of $GInf$ and define $\iota \in GRed_{\mathrm{I}}(N)$ if $\{D \in N \mid D \prec mprem(\iota)\} \cup sprems(\iota) \models concl(\iota)$.

The *ground instances* of a nonground clause C is the set of all ground clauses of the form $C\theta$, where the Boolean subterms of $x\theta$ are included in $\{\top, \bot\}$ for all free variables of C. Let $\mathcal{G}(C)$ denote the set of ground instances of a clause C and if N is a clause set, let $\mathcal{G}(N)$ denote $\bigcup_{C \in N} \mathcal{G}(C)$. The set of ground instances of an inference $\iota \in FInf$ are all inferences $\iota' \in GInf$ such that

- both inferences ι and ι' are instances of the same inference rule, or from ∀Hoist and G∀Hoist, or ∃Hoist and G∃Hoist, or ∀Rw and GForallRw, or ∃Rw and GExistsRw, respectively; and
- for a grounding substitution θ it holds that $prems(\iota') = prems(\iota)\theta$ and $concl(\iota') = concl(\iota)\theta$; and
- the corresponding ground literals and ground Boolean subterms are selected in ι' as in ι after applying θ.

As for clauses, let $\mathcal{G}(\iota)$ denote the set of all ground instances of an inference ι.

Let C be a nonground clause and N be a nonground clause set. The set of redundant clauses with respect to N is denoted by $FRed_{\mathrm{C}}(N)$. Let $C \in FRed_{\mathrm{C}}(N)$ if C is strictly subsumed by a clause in N or $\mathcal{G}(C) \subseteq GRed_{\mathrm{C}}(\mathcal{G}(N))$. Let ι be a nonground inference of $FInf$. The set of redundant inferences w.r.t. clause set N is denoted by $FRed_{\mathrm{I}}(N)$ and define $\iota \in FRed_{\mathrm{I}}(N)$ if $\mathcal{G}(\iota) \subseteq GRed_{\mathrm{I}}(\mathcal{G}(N))$. Finally, let the redundancy criterion for the nonground inference system $FInf$ be $FRed = (FRed_{\mathrm{C}}, FRed_{\mathrm{I}})$.

4.1.5 Refutational Completeness

A usual approach for proving the dynamic completeness of saturation-based calculus is to first prove static completeness of the ground calculus and then obtain dynamic completeness for the non-ground calculus by lifting using the saturation framework, as described in Chapter 3. The static completeness of the ground calculus is proved as by Bachmair and Ganzinger [BG01]. The general proof idea is to construct a model for a saturated set of clauses N with $\bot \notin N$. From the clause set N a term rewrite system is constructed which can be used as a model of N. To view a term rewrite system as an interpretation for first-order logic with interpreted Booleans the next definition is necessary.

Definition 7 (Interpretable Rewrite System)

A rewrite system R over ground terms is *interpretable* if the following conditions are satisfied:

(1) $t \leftrightarrow_R^* \top$ or $t \leftrightarrow_R^* \bot$ for all ground Boolean terms;

(2) for all ground Boolean terms t:

$$\begin{aligned} \neg\top &\leftrightarrow_R^* \bot & \qquad \neg\bot &\leftrightarrow_R^* \top \\ \top \wedge t &\leftrightarrow_R^* t & \qquad t \wedge \top &\leftrightarrow_R^* t \\ \bot \wedge t &\leftrightarrow_R^* \bot & \qquad t \wedge \bot &\leftrightarrow_R^* \bot \\ \bot \vee t &\leftrightarrow_R^* t & \qquad t \vee \bot &\leftrightarrow_R^* t \\ \top \vee t &\leftrightarrow_R^* \top & \qquad t \vee \top &\leftrightarrow_R^* \top \\ \top \to t &\leftrightarrow_R^* t & \qquad t \to \top &\leftrightarrow_R^* \top \\ \bot \to t &\leftrightarrow_R^* \top & \qquad t \to \bot &\leftrightarrow_R^* \neg t \end{aligned}$$

(3) $t \approx t' \leftrightarrow_R^* \top$ if and only if $t \leftrightarrow_R^* t'$ for all ground Boolean terms t and t';

(4) $t \not\approx t' \leftrightarrow_R^* \bot$ if and only if $t \leftrightarrow_R^* t'$ for all ground Boolean terms t and t';

(5) $\forall x.\, t \leftrightarrow_R^* \top$ if and only if $t\{x \mapsto u\} \leftrightarrow_R^* \top$ for every ground Boolean term u;

(6) $\exists x.\, t \leftrightarrow_R^* \bot$ if and only if $t\{x \mapsto u\} \leftrightarrow_R^* \bot$ for every ground Boolean term u.

Given an interpretable term rewrite system R an interpretation $(\mathcal{U}, \mathcal{J})$ can be defined. When an interpretation $(\mathcal{U}, \mathcal{J})$ is constructed from a term rewrite system R it is common to also write R when referring to the interpretation. For a type τ the universe $\mathcal{U}_\tau$ consists of the set of all equivalence classes of terms t of type τ modulo $\leftrightarrow_R^*$. For a term t, denote with $[t]$ its equivalence class modulo $\leftrightarrow_R^*$, i.e., $[t] = \{t' \mid t' \leftrightarrow_R^* t\}$. Let $\mathcal{J}(\mathsf{f}\langle\bar{\tau}_n\rangle)(\bar{a}) = [\mathsf{f}\langle\bar{\tau}_n\rangle(\bar{t})]$. Here, every argument a_i is an equivalence class of terms and we let each t_i be an arbitrary term of the equivalence class a_i. It does not matter what term t_i is chosen, since if another term t_i' was chosen instead, it would still hold that $t_i \leftrightarrow_R^* t_i'$ and therefore also $\mathsf{f}\langle\bar{\tau}_n\rangle(t_1, \ldots, t_i, \ldots, t_n) \leftrightarrow_R^* \mathsf{f}\langle\bar{\tau}_n\rangle(t_1, \ldots, t_i', \ldots, t_n)$. The set of terms $[\top]$ is identified with 1 and $[\bot]$ is identified with 0. Because of requirement (1) of Definition 7 it follows that $\mathcal{U}_o = \{0, 1\}$, $\mathcal{J}(\top) = 1$, and $\mathcal{J}(\bot) = 0$, as required by (I1) and (I2). Requirements (2), (3), and (4) guarantee that conditions (I3) to (I8) for interpretations are satisfied. The last requirements (5) and (6) ensure that quantified terms are interpreted as expected.

Lemma 1 ([Num+21, Lemma 17])

For all ground terms t and interpretable term rewrite systems R it holds that $[\![t]\!]_R = [t]$.

Lemma 1 shows the fact that $R \models t \equiv t'$ if and only if $t \leftrightarrow_R^* t'$, or equivalently $[t] = [t']$.

Having constructed an interpretable term rewrite system R_N from a saturated set of clauses N with $\bot \notin N$ it remains to show that R is actually a model of N. This is done using the Bachmair and Ganzinger's general approach of *reducing counterexamples* [BG01, Section 4.2]. R is called the *candidate model* and a clause $C \in N$ such that $R \not\models C$ is a *counterexample*. C is a *minimal* counterexample if there is no clause $D \in N$ such that $C \succ D$ and $R \not\models D$. If a counterexample C is the main premise of an inference, the side premises are true in R and the conclusion of the inference is a smaller counterexample w.r.t. $\succ$, then the inference is said to *reduce* the counterexample C. If it holds that for every set of clauses N with minimal counterexample C, there is an inference using clauses in N that reduces C, the inference system enjoys the *reduction property for counterexamples*. Having proven that an inference system has the reduction property for counterexamples, in this case *GInf*, the static refutational completeness can be obtained by using Theorem 4.9 of Bachmair and Ganzinger's framework [BG01].

The result of static refutational completeness for the ground inference system can then be used to show dynamic refutational completeness for the non-ground inference system by lifting the inferences and employing the saturation framework by Waldman et al. [Wal+22].

Theorem 2 ([Num+21])
The calculus $(FInf, FRed)$ is dynamically refutationally complete.

4.1.6 Saturation Procedure

When implementing a saturation-based calculus in an automated theorem prover, a *saturation procedure* is needed that infers new clauses from the input problem until either the empty clause is encountered or no irredundant clauses can be inferred anymore. If the calculus is proven to be refutationally complete, the concrete saturation procedure must be able to derive the empty clause in a finite amount of time for unsatisfiable input problems. Here, I will present the *given clause* saturation algorithm, which is used in leading theorem provers such as Vampire [KV13] and E [SCV19]. The Zipperposition theorem prover implements this procedure for first-order logic and uses a generalization for higher-order logic, as will be discussed in sections 4.2.8 and 5.8.

Algorithm 1 shows how the given clause procedure operates in detail.

Algorithm 1 (First-Order Given Clause Algorithm)

Input: Initial set of clauses S
Output: Sat, Unsat, or Unknown

```
1  P ← S
2  A ← ∅
3  while time limit not exceeded
4      if P = ∅
5          return Sat
6      select a clause C from P
7      P ← P \ {C}
8      simplify C by clauses in A
9      if C is trivial or subsumed by A
10         continue
11     if C = ⊥
12         return Unsat
13     simplify clauses in A using C and transfer changed clauses into P
14     A ← A ∪ {C}
15     perform all inferences between C and A and put resulting clauses into P
16 return Unknown
```

Since the unsatisfiability problem for first-order logic is only semi-decidable, there exist inputs for which the given clause algorithm does not terminate with a hypothetical infinite time limit. This may happen, for example, if the saturated set of clauses is infinite. In practice, a time limit is used that dictates when to stop the saturation procedure, as used in line 3 of Algorithm 1.

The algorithm maintains two sets of clauses A and P, which are called *active* and *passive*, respectively. Initially, the input clauses are put into the passive set and the active set is the empty set. In each loop iteration, it is first checked if the passive set is empty. If this is the case, the empty clause $\bot$ was not derived and the result Sat can be returned, which indicates that the input problem is satisfiable. Otherwise, a clause C is selected

from P using some heuristics such as the age of the clause, i.e., in which iteration the clause was added to P, or a notion of weight depending on the syntactical size, the number of variables, and various other properties of clauses. Clause C is also called the given clause. Then, C is removed from P and is tried to be simplified using all the clauses in the active set. In this step, various simplification rules such as subsumption, demodulation, or deletion of trivial literals can be used. Employing strong simplification rules is often crucial for restricting the search space. If C is either trivial or is subsumed by a clause in the active set, the clause is simply discarded and the next iteration is started. Next, it is checked whether C is the empty clause $\bot$. If this is the case, the result Unsat can be reported, which indicates that the input problem is unsatisfiable. This can be justified by showing that all inference rules of the calculus are sound or at least satisfiability preserving. If C is not the empty clause, all clauses in the active set are tried to be simplified or deleted using C and all resulting clauses are transferred to the passive set. The given clause is then added to the active set and all the resulting clauses of all inferences between the given clause and the active set are put into the passive set.

Using the distinction of active and passive sets, the given clause algorithm maintains the invariant that at the beginning of every iteration, all possible inferences between clauses in A have been already performed. Thus, *fairness* is ensured, in the sense that all possible inferences were carried out when a saturated set of clauses is encountered. If this were not the case, the algorithm may report Sat even if there are still inferences that could produce $\bot$.

4.2 Extension to Higher-order Logic

The development of the presented superposition calculus for higher-order logic was split into three milestones, where each milestone builds upon the results of the previous one. The first milestone was to design a calculus for Boolean-free, λ-free higher-order logic [Ben+18]. This logic supports partial application of function symbols, i.e., both g or $\mathsf{g}\ \mathsf{a}$ may be written for a binary function symbol g. Moreover, the type of variables may now be functional and thus variables can occur applied, i.e., $x\ \mathsf{a}$.

Afterwards, a calculus was proposed for Boolean-free higher-order logic with support for λ-expressions [Ben+21]. In this logic, there is no native Boolean type. Instead, the Boolean type needs to be axiomatized. For this, a nullary type constructor *bool* is added to the signature together with symbols $\mathsf{t}, \mathsf{f} : \mathit{bool}$, $\mathsf{not} : \mathit{bool} \to \mathit{bool}$, $\mathsf{and}, \mathsf{or}, \mathsf{impl}, \mathsf{equiv} : \mathit{bool} \to \mathit{bool} \to \mathit{bool}$, $\mathsf{forall}, \mathsf{exists} : \Pi\alpha.\ (\alpha \to \mathit{bool}) \to \mathit{bool}$, $\mathsf{eq} : \Pi\alpha.\ \alpha \to \alpha \to \mathit{bool}$, and $\mathsf{choice} : \Pi\alpha.\ (\alpha \to \mathit{bool}) \to \alpha$. Using these symbols, the Boolean type is characterized using the following axioms:

$\mathsf{t} \not\approx \mathsf{f}$	$\mathsf{impl}\ \mathsf{f}\ x \approx \mathsf{t}$
$x \approx \mathsf{t} \vee x \approx \mathsf{f}$	$x \not\approx y \vee \mathsf{eq}\langle\alpha\rangle\ x\ y \approx \mathsf{t}$
$\mathsf{not}\ \mathsf{t} \approx \mathsf{f}$	$x \approx y \vee \mathsf{eq}\langle\alpha\rangle\ x\ y \approx \mathsf{f}$
$\mathsf{not}\ \mathsf{f} \approx \mathsf{t}$	$\mathsf{equiv}\ x\ y \approx \mathsf{and}\ (\mathsf{impl}\ x\ y)\ (\mathsf{impl}\ y\ x)$
$\mathsf{and}\ \mathsf{t}\ x \approx x$	$\mathsf{forall}\langle\alpha\rangle\ (\lambda x.\ \mathsf{t}) \approx \mathsf{t}$
$\mathsf{and}\ x\ \mathsf{t} \approx x$	$y \approx (\lambda x.\ \mathsf{t}) \vee \mathsf{forall}\langle\alpha\rangle\ y \approx \mathsf{f}$
$\mathsf{or}\ \mathsf{t}\ x \approx \mathsf{t}$	$\mathsf{exists}\langle\alpha\rangle\ y \approx \mathsf{not}\ (\mathsf{forall}\langle\alpha\rangle\ (\lambda x.\ \mathsf{not}\ (y\ x)))$
$\mathsf{or}\ \mathsf{f}\ x \approx x$	$y\ x \approx \mathsf{f} \vee y\ (\mathsf{choice}\langle\alpha\rangle\ y) \approx \mathsf{t}$
$\mathsf{impl}\ \mathsf{t}\ x \approx x$	

Lastly, support for interpreted Booleans was added in order to define a refutationally complete calculus for higher-order logic [Ben+23b], as introduced in Section 2.5. Because the calculus uses dedicated inference and simplification rules to handle Booleans, the axiomatization shown above is no longer necessary and the logical symbols $\wedge, \vee$, etc. replace the symbols $\mathsf{and}, \mathsf{or}$, etc.

For every milestone, the respective calculus was proven to be refutationally complete w.r.t. the used logic and also implemented in the Zipperposition automated theorem prover [Cru15]. Moreover, the overall goal was to take what works best for first-order logic and try to extend this in a *graceful* way. Here, gracefulness means that a prover should only use higher-order reasoning for problems that are actually higher-order. A good and short overview of the milestones and the respective challenges is given in [Bla23]. For an in-depth treatment I refer to the PhD thesis of Alexander Bentkamp [Ben21].

Before introducing the calculus and sketching the completeness proof, I will comment on the several difficulties that arise when moving from first-order to higher-order logic. An interested reader may wonder how it is possible to have a refutationally complete calculus for higher-order logic under the presence of Gödel's first incompleteness theorem. Since arithmetic can be fully formalized in higher-order logic using Peano arithmetic, there must exist a formula F without free variables that can not be proven valid nor proven unsatisfiable. But F must be either valid or unsatisfiable, which contradicts the claim that the presented calculus is refutationally complete. To clear up this dilemma, note how interpretations are defined for functions. That is, $\mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{D}_1, \mathcal{D}_1)$ has to be a subset of the function space from $\mathcal{D}_1$ to $\mathcal{D}_2$ for all $\mathcal{D}_1, \mathcal{D}_2 \in \mathcal{U}$, where $\mathcal{U}$ is the universe of an interpretation.

Gödel's first incompleteness assumes that in every interpretation the set $\mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{D}_1, \mathcal{D}_1)$ is the *full* function space from $\mathcal{D}_1$ to $\mathcal{D}_2$. This kind of semantics is called *standard*. Consider the axiom of induction of Peano arithmetic that can be formalized as follows in higher-order logic, where the type of a unary predicate over natural numbers is simply a function that takes a natural number and results in a Boolean value. Let ω be a 0-ary type constructor for the set of natural numbers and let $\mathsf{zero} : \omega$ be a constant representing

0 and let $\mathsf{succ} : \omega \to \omega$ be the successor function.

$$\forall\langle\omega \to o\rangle\,(\lambda x.\,(x\;\mathsf{zero} \wedge \forall\langle\omega\rangle(\lambda n.\,x\;n \rightarrow x\;(\mathsf{succ}\;n))) \rightarrow \forall\langle\omega\rangle(\lambda n.\,x\;n))$$

In standard semantics, the outermost $\forall$ quantifies over *all* predicates on natural numbers. To obtain a semantics that admits a complete proof system, the logician Leon Henkin introduced the notion of what he called *general models*, but they are usually called Henkin models [Fit02; Hen50]. The idea is that the function space must not be full, but rather can be chosen by proper interpretations such that it is *big enough* in order that λ-expressions are interpreted as expected.

Since it is the case that every standard model is also a Henkin model, if a formula is proven to be valid in Henkin semantics the formula is also valid in standard semantics. A downside to Henkin semantics is that one can not guarantee that quantifiers use the whole function space. Thus, for example, properties for natural numbers are only provable in standard semantics but not in Henkin semantics, since in the latter there are simply more possible interpretations that can act as counterexamples.

To sum up, using nonstandard Henkin semantics it is possible to have a refutationally complete calculus for higher-order logic all of whose derivable formulas are also valid under standard semantics.

4.2.1 Higher-order Unification

As in first-order calculi like resolution or superposition, it is necessary to compute unifiers in the presented higher-order superposition calculus. Unifiers are defined as usual, where a substitution θ is a unifier for two terms s and t if $s\theta$ is equal to $t\theta$. A unification problem is a multiset of unordered term pairs of the same type, written as $\{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$. An element $s_1 \stackrel{?}{=} t_1$ is called *unification constraint*. A substitution is a solution of a unification problem if it is a unifier of every contained unification constraint. More formally, a substitution θ is a solution of $\{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$ if and only if $s_i{\downarrow_{\beta\eta}}$ and $t_i{\downarrow_{\beta\eta}}$ are syntactically equal for every $1 \leq i \leq n$.

Using a reduction from Hilbert's tenth problem, it was proven by Goldfarb that already second-order unification is undecidable [Gol81]. Hilbert's tenth problem asks whether it is possible to devise an algorithm that decides if a given polynomial with integer coefficients has an integer solution. This problem was posed by Hilbert in 1900 and was proven to be undecidable in 1970 [Coo03].

In n-th order unification, every term has to be of order at most n w.r.t. the notion of order given in Definition 1. There are several outcomes of unification problems. Consider the problem $\{\mathsf{f}\;x \stackrel{?}{=} \mathsf{c}\}$, which appropriate types. Clearly, there are no solutions, since x is the only variable and no matter what term x is replaced with, the heads of the terms remain different. Next, the problem $\{x\;\mathsf{c} \stackrel{?}{=} y\}$ has an infinite set of solutions of the form $\{x \mapsto \lambda z.\,t, y \mapsto t\}$, where t is a term with $z \notin \mathcal{FV}(t)$. However, it admits a most general unifier $\{x \mapsto \lambda z.\,y\}$.

Lastly, we consider two problems from [Pre98]. The problem $\{x\,\mathsf{a} \stackrel{?}{=} \mathsf{a}\}$ has exactly two solutions that are incomparable w.r.t. generality: $\{x \mapsto \lambda z.\, z\}$ and $\{x \mapsto \lambda z.\, \mathsf{a}\}$. The remaining case is a problem with an infinite number of incomparable solutions: $\{x\,(\mathsf{f}\mathsf{a}) \stackrel{?}{=} \mathsf{f}(x\,\mathsf{a})\}$. Every solution has the form $\{x \mapsto \lambda z.\, \mathsf{f}^n\,\mathsf{a}\}$, where $n \geq 0$ and f^n denotes the n-fold application of f.

A *complete set of unifiers*, abbreviated as CSU, for two terms s and t on a set of variables X, denoted by $\mathrm{CSU}_X(s,t)$, is an arbitrary set of unifiers of s and t such that if θ is a unifier of s and t there exists some $\sigma \in \mathrm{CSU}_X(s,t)$ and a substitution ρ such that $x\sigma\rho = x\theta$ for all variables $x \in X$. It can be assumed that every substitution of $\mathrm{CSU}_X(s,t)$ is idempotent. Usually, X will be left implicit and will be given by the set of free variables of all the clauses in which the terms s and t occur. Note that a complete set of unifiers may include redundant unifiers. Two terms s and t have a most general unifier if $\mathrm{CSU}_X(s,t)$ is a singleton set.

Although higher-order unification is undecidable, it is semi-decidable. That is, there exist algorithms that enumerate a complete set of unifiers of the given terms. However, such algorithms may not terminate in general, depending on the input.

A prominent algorithm for enumerating CSUs is Jensen and Pietrzykowski's (JP) procedure [JP76]. Recently, a refinement of the JP procedure was developed by Vukmirović, Bentkamp and Nummelin that uses decision procedures for decidable fragments of higher-order unification as well as restrictions of the search space to avoid enumerating many redundant unifiers [VBN20]. Decidable fragments of higher-order unification are also called *oracles*. In Zipperposition, a complete version of the algorithm, as well as a pragmatic version was implemented, that limits the search space to obtain a terminating but incomplete algorithm. Moreover, the JP procedure is implemented.

4.2.2 Preprocessing

In the higher-order superposition calculus, several inferences are restricted to subterms that resemble first-order subterms. To this end, *green subterms* are introduced.

Definition 8 (Green subterms and positions)

A *green position* of a λ-term is a finite sequence of natural numbers that is inductively defined as follows. The empty sequence ε is a green position of any λ-term. For every symbol $\mathsf{f} \in \Sigma \setminus \{\forall, \exists\}$, types $\bar{\tau}$, and λ-terms $\bar{u}$, if p is a green position of u_i, then $i.p$ is a green position of $\mathsf{f}\,\bar{u}$.

Then *green subterm* of a λ-term at a green position is defined inductively as follows. For all λ-terms t and green position ε, t is the greeun subterm of t at green position ε. For all symbols $\mathsf{f} \in \Sigma \setminus \{\forall, \exists\}$, types $\bar{\tau}$, and λ-terms $\bar{u}$, if t is a green subterm of u_i at some green position p for some i, then t is the green subterm of $\mathsf{f}\,\bar{u}$ at green position $i.p$.

To refer to green positions in clauses and literals, a sequence of natural numbers cannot be used since clauses and literals are unordered. To this end, a green position in a clause C is defined to be a tuple of the form $L.s.p$, where $L = s \mathrel{\dot{\approx}} t$ is a literal in C and p is a green position in s. The green subterm of C at green position $L.s.p$ is the green subterm of s at green position p. A green position $L.s.p$ is *top level* if $p = \varepsilon$. If p is a green position of s, we denote the green subterm at position p in s with $s|_p$. The notions of *at or below* and *below* are defined as in Section 2.2.1.

Consider the term $t = \mathsf{f}\ (\neg\mathsf{a})\ (\forall\langle\tau\rangle\ (\lambda x.\ \mathsf{p}\ x))\ (z\ \mathsf{b})\ (\lambda x.\ \mathsf{c}\ \mathsf{d})$. The list of green subterms of t are enumerated as follows. First of all, t itself is a green subterm of t at position ε. because $\mathsf{f} \notin \{\forall, \exists\}$, the arguments of f are checked. Thus, $\neg\mathsf{a}$ is a subterm of t at position 1, $\forall\langle\tau\rangle\ (\lambda x.\ \mathsf{p}\ x)$ is a subterm of t at position 2, $z\ \mathsf{b}$ is a subterm of t at position 3, and $\lambda x.\ \mathsf{c}\ \mathsf{d}$ is a subterm of t at position 4. Continuing the recursive definition, it follows that a is a green subterm of t at position 1.1 because $\neg\mathsf{a}$ is a green subterm of t at position 1 and $\neg \notin \{\forall, \exists\}$. Since the term $\forall\langle\tau\rangle\ (\lambda x.\ \mathsf{p}\ x)$ is headed by a quantifier, no proper green subterms of this term can be green subterms. For $z\ \mathsf{b}$ there are also no proper green subterms, because the term is headed by a variable. Lastly, for $\lambda x.\ \mathsf{c}\ \mathsf{d}$ there are also no proper green subterms because the term is a λ-abstraction. Hence, a green subterm cannot occur in a *lambda*-abstraction, applied to a variable, or inside a quantifier $\forall$ or $\exists$.

Definition 9 (Green contexts)

Let $s\langle u\rangle_p$ denote a λ-term s with the green subterm u at position p. $s\langle\rangle_p$ is called a *green context*, where p may be omitted. Moreover, let $C\langle u\rangle_p$ denote a clause C with the green subterm u at position $p = L.s.q$ for $L = s \mathrel{\dot{\approx}} t$, $L \in C$, and $s\langle u\rangle_q$.

Green positions, subterms, and contexts are lifted to $\beta\eta$-equivalence classes via the $\beta\eta\mathsf{Q}_\eta$-normal representative.

The addition of interpreted quantifiers introduced some difficulties because quantified variables that appear in higher-order contexts cannot be translated to first-order logic as in previous milestones. However, a preprocessing step can be used to eliminate all such problematic occurrences of variables.

Definition 10 (Preprocessing via rewrite rules $\forall_\approx$ and $\exists_\approx$)

The rewrite rules $\forall_\approx$ and $\exists_\approx$ are given by

$$\forall\langle\tau\rangle \longrightarrow_{\forall_\approx} \lambda y.\ y \mathrel{\approx} (\lambda x.\ \top)$$
$$\exists\langle\tau\rangle \longrightarrow_{\exists_\approx} \lambda y.\ y \not\approx (\lambda x.\ \bot)$$

where the rewritten occurrence $\mathsf{Q}\langle\tau\rangle$ is either unapplied, has an argument that is not a λ-expression, or has an argument of the form $\lambda x.\ v$ such that x occurs free in a nongreen position of v. Both rewrite rules $\forall_\approx$ and $\exists_\approx$ are collectively denoted by $\mathsf{Q}_\approx$. If one of these rules can be applied to a term, the term is called $\mathsf{Q}_\approx$-*reducible*. Otherwise, it is called $\mathsf{Q}_\approx$-*normal*. A clause set is $\mathsf{Q}_\approx$-normal if all terms occurring in the set are $\mathsf{Q}_\approx$-normal.

The initial clause set can be preprocessed to yield a $\mathsf{Q}_{\approx}$-normal clause set. This procedure does terminate, because the number of quantifiers is reduced by every application of a rewrite rule of $\mathsf{Q}_{\approx}$.

4.2.3 Term Orders

The term orders for the calculus have to obey several restrictions, which are defined next.

Definition 11 (Strict Ground Term Order)
Let $\succ$ be a well-founded strict total order on ground terms. It is a *strict ground term order* if the following conditions are satisfied:

(O1) compatibility with green contexts: $s' \succ s$ implies $t\langle s' \rangle \succ t\langle s \rangle$;

(O2) green subterm property: $t\langle s \rangle \succeq s$;

(O3) $u \succ \bot \succ \top$ for every term $u \notin \{\top, \bot\}$;

(O4) $\mathsf{Q}\langle\tau\rangle\, t \succ t\, u$ for every type t, term t and u such that $\mathsf{Q}\langle\tau\rangle\, t$ and u are $\mathsf{Q}_{\approx}$-normal and the only Boolean green subterms of u are $\top$ and $\bot$.

A strict ground term order is extended to literals and clauses using the multiset order, as explained in Section 2.1. The condition (O4) cannot be stated for terms t and u without the restrictions. If t and u were allowed to be regular terms, it would be the case that $\mathsf{Q}\langle\tau\rangle\, (\lambda x.\, x) \succ ((\lambda x.\, x)\, \mathsf{Q}\langle\tau\rangle\, (\lambda x.\, x))$, where the term $\lambda x.\, x$ is used for t and u. Applying a β-reduction to the right-hand side yields $((\lambda x.\, x)\, \mathsf{Q}\langle\tau\rangle\, (\lambda x.\, x)) = \mathsf{Q}\langle\tau\rangle\, (\lambda x.\, x)$, which would imply that $\mathsf{Q}\langle\tau\rangle\, (\lambda x.\, x) \succ \mathsf{Q}\langle\tau\rangle\, (\lambda x.\, x)$. But this contradicts the assumption that the strict $\succ$ is irreflexive. This issue is fixed by letting the Boolean green subterms of t and u to only be among $\{\top, \bot\}$. But there is also a further issue:

$$\mathsf{Q}\langle\tau\rangle\, (\lambda y.\, y\, \mathsf{a}) \succ (\lambda y.\, y\, \mathsf{a})\, (\lambda x.\, \mathsf{Q}\langle\tau\rangle\, (\lambda y.\, y\, \mathsf{a})) = \mathsf{Q}\langle\tau\rangle\, (\lambda y.\, y\, \mathsf{a})$$

This again contradicts the assumptions on $\succ$ and the restrictions regarding Boolean green subterms does not solve this issue. To circumvent this problem, $\mathsf{Q}_{\approx}$-normality is needed.

Definition 12 (Strict and Nonstrict Term Orders)
A binary relation $\succ$ on terms, literals and clauses is a *strict term order* if it is a strict ground term order when restricted to ground terms and it is stable under grounding substitutions, that is $t \succ s$ implies that $t\theta \succ s\theta$ for all substitutions θ that ground s and t.

Let $\succ$ be a strict term order. A *nonstrict term order* is a binary relation $\succsim$ on terms, literals and clauses such that $t \succsim s$ implies $t\theta \succeq s\theta$ for all substitutions θ that ground s and t.

A reason to use a nonstrict term order $\succsim$ instead of the reflexive closure $\succeq$ of a strict term order $\succ$ is that using $\succsim$ more terms can be compared. Consider again the terms $x\, \mathsf{a}$

and $x\ \mathsf{b}$. It cannot be the case that $x\ \mathsf{a} \succeq x\ \mathsf{b}$ because the terms are not syntactically equal and if $\theta = \{x \mapsto \lambda y.\ \mathsf{c}\}$ it cannot hold that $x\ \mathsf{a} \succ x\ \mathsf{b}$ by stability w.r.t. grounding substitutions. But using $\succsim$ it is possible to have that $x\ \mathsf{a} \succsim x\ \mathsf{b}$ when $\mathsf{a} \succ \mathsf{b}$.

In [Ben+23b, section 3.9] a concrete term order fulfilling properties of a strict term order is constructed via a translation into untyped first-order logic and the transfinite Knuth-Bendix order [LW07].

4.2.4 Selection Functions and Eligibility

As in the first-order case, the calculus is parameterized by two selection functions and needs a notion of eligibility to specify the possible positions, where inferences are allowed to happen. The corresponding definitions are given below.

Definition 13 (Selection Functions)

A literal selection function is a mapping from a clause to a subset of its negative literals with the restriction that a literal $L\langle y\rangle$ must not be selected if $y\ \bar{t}_n$, where $n \geq 1$, is a $\succeq$-maximal term of the respective clause.

A boolean subterm selection function maps a clause C to a subset of its green positions with Boolean subterms. These positions and subterms at these positions are called selected in C. A subterm s must not be selected if

- a variable y is a green subterm of s and $y\ \bar{t}_n$, where $n \geq 1$, is a $\succeq$-maximal term of the respective clause; or
- $s \in \{\mathsf{T}, \bot\}$ or it is variable-headed; or
- s is at the top-level position on either side of a positive literal.

Definition 14 (Eligibility)

A literal L is (*strictly*) $\trianglerighteq$-*eligible* w.r.t. a substitution θ in clause C for some binary relation $\trianglerighteq$ if it is selected in C or there are no selected literals and no selected Boolean subterms in C and $L\theta$ is (strictly) $\trianglerighteq$-maximal in $C\theta$.

The $\trianglerighteq$-eligible positions of a clause C w.r.t. a substitution θ are defined as follows:

- All selected positions are $\trianglerighteq$-eligible.
- If a literal $L = s \mathrel{\tilde{\approx}} t$ with $s\theta \not\trianglelefteq t\theta$ is either $\trianglerighteq$-eligible and negative or strictly $\trianglerighteq$-eligible and positive, then $L.s.\varepsilon$ is $\triangleright$-eligible.
- If position p is $\trianglerighteq$-eligible and the head of $C|_p$ is not $\approx$ or $\not\approx$, the positions of all direct green subterms are $\trianglerighteq$-eligible.
- If position p is $\trianglerighteq$-eligible and $C|_p$ is of the form $s \approx t$ or $s \not\approx t$, then the position of s is $\trianglerighteq$-eligible if $s\theta \not\trianglelefteq t\theta$ and the position of t is $\trianglerighteq$-eligible if $s\theta \not\trianglerighteq t\theta$.

4.2.5 Inference Rules

The inference rules of the calculus depend on the notions of *deep occurences* of variables and *fluid terms*, that are defined next.

Definition 15 (Deep Occurrences)
A variable is said to *occur deeply* in a clause C if it occurs as a subterm of an argument of a variable or as a subterm of a λ-expression that is not directly below a quantifier.

The notion of deep variable occurrences is needed to characterize variables in a clause C with an occurrence at a position inside a λ-expression in ground instances of C. In several inference rules it is forbidden to rewrite a deeply occurring variable or a fluid term.

Definition 16 (Fluid Terms)
A term t is *fluid* if one of the following is true:

- $t{\downarrow_{\beta\eta\mathsf{Q}_\eta}}$ is of the form $x\,\bar{t}_n$ with $n \geq 1$; or
- $t{\downarrow_{\beta\eta\mathsf{Q}_\eta}}$ is a λ-expression and there exists a substitution θ such that $t\theta{\downarrow_{\beta\eta\mathsf{Q}_\eta}}$ is not a λ-expression because of η-reduction.

Fluid terms can change their form fundamentally under substitutions. Hence, the FluidSup rule introduced below is needed to deal with variables occurring deeply and with fluid terms.

$$\frac{\overbrace{D' \vee t \approx t'}^{D} \qquad C\langle u\rangle}{(D' \vee C\langle t'\rangle)\sigma}\ \textsc{Sup}$$

1. u is not fluid;
2. u is not a variable occurring deeply in C;
3. *variable condition*: if u is a variable x, there must exist a grounding substitution θ such that $t\sigma\theta \succ t'\sigma\theta$ and $C\sigma\theta \prec (C\{x \mapsto t'\})\sigma\theta$;
4. $\sigma \in \mathrm{CSU}(t, u)$;
5. $t\sigma \not\precsim t'\sigma$;
6. the position of u is $\succsim$-eligible in C w.r.t. σ;
7. $C\sigma \not\precsim D\sigma$;
8. $t \approx t'$ is strictly $\succsim$-eligible in D w.r.t. σ;
9. $t\sigma$ is not a fully applied logical symbol;
10. if $t'\sigma = \bot$, the position of the subterm u is at the top level of a positive literal.

$$\frac{\overbrace{D' \vee t \approx t'}^{D} \qquad C\langle u \rangle}{(D' \vee C\langle z\ t' \rangle)\sigma}\ \textsc{FluidSup}$$

1. u is a variable occurring deeply in C or u is a fluid term;
2. z is a fresh variable;
3. $\sigma \in \mathrm{CSU}(z\ t, u)$;
4. $(z\ t')\sigma \neq (z\ t)\sigma$;
5. - 10. conditions 5 to 10 from Sup.

$$\frac{\overbrace{C' \vee u \not\approx u'}^{C}}{C'\sigma}\ \textsc{ERes}$$

1. $\sigma \in CSU(u, u')$;
2. $u \not\approx u'$ is $\succsim$-eligible in C w.r.t.. σ.

$$\frac{\overbrace{C' \vee u' \approx v' \vee u \approx v}^{C}}{(C' \vee v \not\approx v' \vee u \not\approx v')\sigma}\ \textsc{EFact}$$

1. $\sigma \in \mathrm{CSU}(u, u')$;
2. $u\sigma \not\precsim v\sigma$;
3. $(u \approx v)\sigma$ is $\succsim$-maximal in $C\sigma$;
4. nothing is selected in C.

$$\frac{\overbrace{C' \vee s \approx s'}^{C}}{C'\sigma \vee s\sigma\ \bar{x}_n \approx s'\sigma\ \bar{x}_n}\ \textsc{ArgCong}$$

1. $n > 0$;
2. σ is the most general type substitution that ensures well-typedness of the conclusion;
3. $s \approx s'$ is strictly $\succsim$-eligible in C w.r.t. σ;
4. $\bar{x}_n$ is a tuple of distinct fresh variables.

$$\frac{C\langle u\rangle}{(C\langle \bot\rangle \vee u \approx \top)\sigma}\ \textsc{BoolHoist}$$

1. the type of u is either the Boolean type o or $\Pi\alpha.\ \alpha$ for some type variable α;
2. if u is of Boolean type, then $\sigma = \mathsf{id}$, otherwise $\sigma = \{\alpha \mapsto o\}$;
3. u is neither variable-headed nor a fully applied logical symbol;
4. the position of u is $\gtrsim$-eligible in C w.r.t. σ;
5. the occurrence of u is not at the top level of a positive literal.

$$\frac{\overbrace{C' \vee s \approx s'}^{C}}{C'\sigma}\ \textsc{FalseElim}$$

1. $\sigma \in \mathrm{CSU}(s \approx s', \bot \approx \top)$;
2. $s \approx s'$ is strictly $\gtrsim$-eligible in C w.r.t. σ.

$$\frac{C\langle u\rangle}{(C\langle \bot\rangle \vee x \approx y)\sigma}\ \textsc{EqHoist} \qquad \frac{C\langle u\rangle}{(C\langle \top\rangle \vee x \approx y)\sigma}\ \textsc{NeqHoist}$$

$$\frac{C\langle u\rangle}{(C\langle \bot\rangle \vee y\ x \approx \top)\sigma}\ \textsc{ForallHoist} \qquad \frac{C\langle u\rangle}{(C\langle \top\rangle \vee y\ x \approx \bot)\sigma}\ \textsc{ExistsHoist}$$

1. $\sigma \in \mathrm{CSU}(u, t)$, where t is
   - $x \mathrel{\approx\!\!\!\approx} y$ for EqHoist;
   - $x \not\approx y$ for NeqHoist;
   - $\forall\langle\alpha\rangle\ y$ for ForallHoist;
   - $\exists\langle\alpha\rangle\ y$ for ExistsHoist.
2. x, y are fresh term variables and α is a fresh type variable;
3. the position of u is $\gtrsim$-eligible in C w.r.t. σ;
4. if the head of u is a variable, it must be applied and the affected literal must be of the form $u \approx \top$, $u \approx \bot$, or $u \approx v$, where v is a variable-headed term.

$$\frac{C\langle u\rangle}{C\langle t'\rangle\sigma}\ \textsc{BoolRw}$$

1. $\sigma \in \mathrm{CSU}(t, u)$;
2. (t, t') is one of the following pairs, where x is a fresh variable:

$(\neg\bot, \top)$	$(\bot \wedge \top, \bot)$	$(\bot \vee \top, \top)$	$(\bot \to \top, \top)$
$(\neg\top, \bot)$	$(\top \wedge \top, \top)$	$(\top \vee \top, \top)$	$(\top \to \top, \top)$
$(\bot \wedge \bot, \bot)$	$(\bot \vee \bot, \bot)$	$(\bot \to \bot, \top)$	$(x \approx x, \top)$
$(\top \wedge \bot, \bot)$	$(\top \vee \bot, \top)$	$(\top \to \bot, \bot)$	$(x \not\approx x, \bot)$

3. u is not a variable;
4. the position of u is $\gtrsim$-eligible in C w.r.t. σ;
5. if the head of u is a variable, it must be applied and the affected literal must be of the form $u \approx \top$, $u \approx \bot$, or $u \approx v$, where v is a variable-headed term.

$$\frac{C\langle u\rangle}{C\langle y\ (\mathsf{sk}_{\Pi\bar{\alpha}.\,\forall\bar{x}.\,\exists z.\,\neg(y\sigma)\ z}\ \langle\bar{\alpha}\rangle\ \bar{x})\rangle\sigma}\ \textsc{ForallRw}$$

$$\frac{C\langle u\rangle}{C\langle y\ (\mathsf{sk}_{\Pi\bar{\alpha}.\,\forall\bar{x}.\,\exists z.\,(y\sigma)\ z}\ \langle\bar{\alpha}\rangle\ \bar{x})\rangle\sigma}\ \textsc{ExistsRw}$$

1. $\sigma \in \mathrm{CSU}(\mathsf{Q}\langle\beta\rangle\ y, u)$, where Q is $\forall$ for ForallRw and is $\exists$ for ExistsRw, β is a fresh type variable, y is a fresh term variable, $\bar{\alpha}$ are the free type variables and $\bar{x}$ are the free term variables occurring in $y\sigma$ in order of first occurrence;
2. u is not a variable;
3. the position of u is $\gtrsim$-eligible in C w.r.t. σ;
4. if the head of u is a variable, it must be applied and the affected literal mus tbe of the form $u \approx \top$, $u \approx \bot$, or $u \approx v$, where v is a variable-headed term;
5. the indicated occurrence of u is not in a literal $u \approx t$, where t is $\top$ for ForallRw and t is $\bot$ for ExistsRw.

$$\frac{C\langle u\rangle}{(C\langle z\ \bot\rangle \vee x \approx \top)\sigma}\ \textsc{FluidBoolHoist}$$

1. u is fluid;
2. z and x are fresh variables;
3. $\sigma \in \mathrm{CSU}(z\ x, u)$;
4. $x\sigma \notin \{\top, \bot\}$;
5. the position of u is $\gtrsim$-eligible in C w.r.t. σ;
6. $(z\ \bot)\sigma \neq (z\ x)\sigma$.

$$\frac{C\langle u\rangle}{(C\langle z\ \top\rangle \vee x \approx \bot)\sigma}\ \textsc{FluidLoobHoist}$$

1.-5. conditions 1 to 5 from FluidBoolHoist;
6. $(z\ \top)\sigma \neq (z\ x)\sigma$.

Besides the inference rules, two axioms are part of the calculus, which are given below.

$$z\ (\mathsf{diff}\langle\alpha,\beta\rangle\ z\ y) \not\approx y\ (\mathsf{diff}\langle\alpha,\beta\rangle\ z\ y) \vee z \approx y \qquad \text{(Ext)}$$

$$y\ x \approx \bot \vee y\ (\varepsilon\langle\alpha\rangle\ y) \approx \top \qquad \text{(Choice)}$$

Axiom Ext ensures functional extensionality, i.e., the property that two functions are equal if they agree on all inputs. Therein, the types of z and y are $\Pi\alpha,\beta.\ \alpha \to \beta$. The function symbol $\mathsf{diff}\langle\alpha,\beta\rangle$ is an abbreviation for $\mathsf{sk}_{\Pi\alpha,\beta.\ \forall z.\forall y.\ \exists x.\ z\ x \not\approx y\ x}\langle\alpha,\beta\rangle$ Axiom Ext can be seen as a skolemized form of $(\mathsf{\forall}\langle\alpha\rangle\ (\lambda x.\ z\ x \approx z\ y)) \rightarrow z \approx y$. Correct semantics of the Hilbert choice operator ε is ensured through axiom Choice.

All of the presented inference rules and axioms are sound w.r.t. $\models\!\!\!\approx$. Moreover, every rule that does not introduce Skolem symbols is sound w.r.t. $\models$ and the preprocessing is sound w.r.t. $\models$ and $\models\!\!\!\approx$. Additionally, every derivation using the inference rules and axioms is satisfiability-preserving w.r.t. $\models$ and $\models\!\!\!\approx$ if the initial clause set does not contain any Skolem symbols.

4.2.6 Redundancy Criterion

The redundancy criterion and the completeness proof of the presented first-level calculus use a ground level to lift redundancy and completeness to the nonground level. In the higher-order calculus we do the same but with three levels: a higher-order level H, a $\mathsf{Q}_{\approx}$-normal ground higher-order level GH and a ground first-order level GF with interpreted Booleans. The logic of the first-order level is the same as the one used in Section 2.2. Let $\mathcal{C}_{\mathrm{H}}$, $\mathcal{C}_{\mathrm{GH}}$, and $\mathcal{C}_{\mathrm{GF}}$ denote the sets of clauses of levels H, GH, and GF, respectively. Likewise, the sets $\mathcal{T}_{\mathrm{H}}$, $\mathcal{T}_{\mathrm{GH}}$, $\mathcal{T}_{\mathrm{GF}}$ denote the respective sets of terms.

Let $(\Sigma_{\mathsf{ty}}, \Sigma)$ be the signature of level H. Level GH has the same signature, but the terms and clauses of this level are $\mathsf{Q}_{\approx}$-normal and ground. The signature of GF $(\Sigma_{\mathsf{ty}}, \Sigma_{\mathrm{GF}})$ is constructed using the signature of level H $(\Sigma_{\mathsf{ty}}, \Sigma)$. The set of type constructors Σ_{ty} are the same with the caveat that $\to$ is an uninterpreted type constructor for GF that must not be equated with the arrow used for type declarations in the logic.

The set of constant and function symbols Σ_{GF} is defined as follows. For every instance $\mathsf{f}\langle\bar{\upsilon}\rangle : \bar{\tau}_n \to \tau$ of a symbol $\mathsf{f} \in \Sigma$, where no type variables occur in $\bar{\upsilon}$, a set of first-order symbols is introduced in the following way. For every $0 \leq j \leq n$, a first-order symbol $\mathsf{f}_j^{\bar{\upsilon}} \in \Sigma_{\mathrm{GF}}$ is introduced with argument types $\tau_1 \times \cdots \times \tau_j$ and result type $\tau_{j+1} \to \cdots \to \tau_n \to \tau$. This procedure is executed for logical and nonlogical symbols. If f takes no type arguments f_j is written instead of $\mathsf{f}_j^{\varepsilon}$.

For example, if g is of type $\kappa \to \gamma \to \varrho$, three first-order symbols will be introduced because g takes no type parameters. The corresponding first-order symbols are $\mathsf{g}_0 : \kappa \to \gamma \to \varrho$, $\mathsf{g}_1 : \gamma \to \varrho$, and $\mathsf{g}_2 : \varrho$. This mechanism is used in the first-order encoding for partial application of function symbols. That is, an occurrence of g that is applied to n arguments will be represented by g_n on the first-order level GF. The symbols $\bot_0$, $\top_0$, $\wedge_2$, $\vee_2$, $\rightarrow_2$, $\approx_2$, $\not\approx_2$ are identified with their corresponding logical symbols on the GF level. The higher-order quantifiers $\forall$ and $\exists$ are not included in this procedure, since they will be translated to ordinary first-order quantifiers $\forall$ and $\exists$. In order to represent λ-expressions, for each ground term $\lambda x.\, t$, a symbol $\mathsf{lam}_{\lambda x.\, t} \in \Sigma_{\mathrm{GF}}$ is introduced of the same type.

The levels H, GH, GF are connected via a grounding function $\mathcal{G}$ and an encoding function $\mathcal{F}$, which looks as follows:

$$\begin{array}{c} \text{H} \\ \text{higher-order level} \end{array} \xrightarrow{\mathcal{G}} \begin{array}{c} \text{GH} \\ \text{ground } \mathsf{Q}_{\approx}\text{-normal} \\ \text{higher-order level} \end{array} \xrightarrow{\mathcal{F}} \begin{array}{c} \text{GF} \\ \text{ground first-order level} \\ \text{with interpreted Booleans} \end{array}$$

The functions $\mathcal{G}$ and $\mathcal{F}$ are defined as follows.

Definition 17 (Grounding function $\mathcal{G}$)

The grounding function $\mathcal{G} : \mathcal{C}_{\mathrm{H}} \to \mathcal{P}(\mathcal{C}_{\mathrm{GH}})$ maps a clause $C \in \mathcal{C}_{\mathrm{H}}$ to its set of *ground instances* $\mathcal{G}(C)$, where every element of $\mathcal{G}(C)$ is of the form $C\theta$ for some grounding substitution θ with the additional property that for all variables x occurring in C the only Boolean green subterms of $x\theta$ are $\top$ and $\bot$.

Definition 18 (Encoding function $\mathcal{F}$)

The function $\mathcal{F} : \mathcal{T}_{\mathrm{GH}} \to \mathcal{T}_{\mathrm{GF}}$ is called an *encoding* function because it encodes properties like amount of argument and type arguments of higher-order terms on the first-order level. The function is recursively defined:

$$\begin{aligned}
\mathcal{F}(\mathsf{f}\langle\bar{\upsilon}\rangle\ \bar{t}_j) &= \mathsf{f}_j^{\bar{\upsilon}}(\mathcal{F}(s_1), \ldots, \mathcal{F}(s_j)) \\
\mathcal{F}(x) &= x \\
\mathcal{F}(\mathsf{\forall}\langle\tau\rangle\ (\lambda x.\ t)) &= \forall x.\ \mathcal{F}(t) \\
\mathcal{F}(\mathsf{\exists}\langle\tau\rangle\ (\lambda x.\ t)) &= \exists x.\ \mathcal{F}(t) \\
\mathcal{F}(\lambda x.\ t) &= \mathsf{lam}_{\lambda x.\ t}
\end{aligned}$$

For every term, the $\downarrow_{\beta\eta\mathsf{Q}_\eta}$ representative is used. The encoding $\mathcal{F}$ is extended to also map clauses in $\mathcal{C}_{\mathrm{GH}}$ to $\mathcal{C}_{\mathrm{GF}}$ by translating each literal and each side of each literal separately.

Because $\mathcal{F}$ is bijective, the order $\succ$ can be transferred from $\mathcal{T}_{\mathrm{GH}}$ to $\mathcal{T}_{\mathrm{GF}}$ and from $\mathcal{C}_{\mathrm{GH}}$ to $\mathcal{C}_{\mathrm{GF}}$ by defining $t \succ s$ as $\mathcal{F}^{-1}(t) \succ \mathcal{F}^{-1}(s)$ for $t, s \in \mathcal{T}_{\mathrm{GF}}$. In the following, $\succ$ is used both for the term-orders of the higher levels and the first-order level. It holds that $\succ$ on $\mathcal{T}_{\mathrm{GF}}$ satisfies all constraints as required by Definition 4.

Now the sets of redundant clauses can be defined. In the following, for a clause set N let $N^{\prec C}$ denote the set of all clauses in N that are smaller w.r.t. $\prec$ than C, i.e., $N^{\prec C} = \{D \in N \mid D \prec C\}$.

Definition 19 (Clause redundancy)

The set of redundant clauses w.r.t. a given set of clauses is defined for each level, where every level builds up on the one that comes before it:

- Let $\mathit{GFRed}_{\mathrm{C}} : \mathcal{P}(\mathcal{C}_{\mathrm{GF}}) \to \mathcal{P}(\mathcal{C}_{\mathrm{GF}})$ be the clause redundancy criterion for level GF, where
$$\mathit{GFRed}_{\mathrm{C}}(N) = \{C \mid N^{\prec C} \models C\}.$$
- Let $\mathit{GHRed}_{\mathrm{C}} : \mathcal{P}(\mathcal{C}_{\mathrm{GH}}) \to \mathcal{P}(\mathcal{C}_{\mathrm{GH}})$ be the clause redundancy criterion for level GH, where
$$\mathit{GHRed}_{\mathrm{C}}(N) = \{C \mid \mathcal{F}(C) \in \mathit{GFRed}_{\mathrm{C}}(\mathcal{F}(N))\}.$$
- Let $\mathit{HRed}_{\mathrm{C}} : \mathcal{P}(\mathcal{C}_{\mathrm{H}}) \to \mathcal{P}(\mathcal{C}_{\mathrm{H}})$ be the clause redundancy criterion for level H, where
$$\begin{aligned}\mathit{HRed}_{\mathrm{C}}(N) = \{C \mid &\text{for all } D \in \mathcal{G}(C) \text{ we have } D \in \mathit{GHRed}_{\mathrm{C}}(\mathcal{G}(N)); \\ &\text{or there exists } C' \in N \text{ such that } C \sqsupset C' \text{ and } D \in \mathcal{G}(C')\}\end{aligned}$$

The binary relation $\sqsupset$ is called a *tiebraker* and can be chosen to be an arbitrary well-founded partial order on $\mathcal{C}_{\mathrm{H}}$.

Each of the three levels has a corresponding inference system, which are denoted by *HInf*, *GHInf*, and *GFInf* for the levels H, GH, and GF, respectively. The system *HInf* consists of the inference rules presented in Section 4.2.5. The selection functions *HLitSel* and *HBoolSel*, with which *HInf* is parameterized, are fixed globally. From these functions, the parameters for *GHInf* are derived as described next.

Definition 20 (Selection functions for level GH)

The literal selection function for the GH level, denoted by *GHLitSel*, maps each clause $C \in \mathcal{C}_{\mathrm{GH}}$ to a subset of the literals occurring in C. The Boolean subterm selection function for the GH level, denoted by *GHBoolSel*, maps each clause $C \in \mathcal{C}_{\mathrm{GH}}$ to a subset of the green positions with Boolean subterms of C. Both functions are required to satisfy the condition that for all $C \in \mathcal{C}_{\mathrm{GH}}$, there is a $D \in \mathcal{C}_{\mathrm{H}}$ with $C \in \mathcal{G}(D)$ such that the selections $\mathit{HLitSel}(D), \mathit{HBoolSel}(D)$ and the selections $\mathit{GHLitSel}(C)$, $\mathit{GHBoolSel}(C)$ correspond.

Similar to the first-order calculus, the GH level is parameterized by a witness function.

Definition 21 (Witness function)

A witness function *GHWit* maps a clause $C \in \mathcal{C}_{\mathrm{GH}}$ and a green position of a quantifier-hreaded term in C to a term $\mathit{GHWit}(C, p) \in \mathcal{T}_{\mathrm{GH}}$ such that $\mathsf{Q}\langle\tau\rangle\, t \succ t\ \mathit{GHWit}(C, p)$ if $C|_p = \mathsf{Q}\langle\tau\rangle\, t$.

The responsibility of the witness function is to provide appropriate Skolem terms that serve as witnesses for the existence of terms that fulfill the concrete predicate. It is not possible to also fix the selection and witness functions for the GH level globally, because they depend on the saturated clause set in the limit of the derivation. Thus, all possible parameters of the GH-level calculus are given as follows.

Definition 22 (Set of parameter triples $\mathcal{Q}$)

A *parameter triple* is a 3-tuple (*GHLitSel*, *GHBoolSel*, *GHWit*) consisting of a GH-level literal selection function *GHLitSel*, a GH-level Boolean subterm selection function *GHBoolSel* and a witness function *GHWit*. Let $\mathcal{Q}$ denote the set of all parameter triples.

The notation GHInf^q, where $q = (\mathit{GHLitSel}, \mathit{GHBoolSel}, \mathit{GHWit})$ stands for the inference system GHInf with parameters q. The inference rules of GHInf^q are:

- Sup
- ERes
- EFact
- BoolHoist
- FalseElim
- EqHoist
- NeqHoist
- BoolRw
- GForallHoist
- GExistsHoist
- GArgCong
- GExt
- GChoice
- GForallRw
- GExistsRw

In the above rules, all references to $\succsim$ are replaced by $\succeq$.

Instead of ForallHoist, ExistsHoist, ArgCong, Ext, Choice, ForallRw, and ExistsRw, the inference system GHInf^q uses the following rules, which are going to be introduced below, because the former rules and axioms use free variables, which is not possible in the ground level GH:

- GForallHoist
- GExistsHoist
- GArgCong
- GExt
- GForallRw
- GExistsRw

Instead, the latter rules need to enumerate ground terms in the conclusions rather than using fresh variables. That is, let $\mathcal{T}^\star_{\mathrm{GH}}$ denote the set of all terms $t \in \mathcal{T}_{\mathrm{GH}}$ such that the set of Boolean green subterms of t is a subset of $\{\mathsf{T}, \mathsf{\bot}\}$. Then, the rules GForallHoist and GExistsHoist are given by:

$$\frac{C\langle \mathsf{\forall}\langle\tau\rangle\ t\rangle_p}{C\langle \mathsf{\bot}\rangle \vee t\ u \approx \mathsf{T}}\ \text{GForallHoist}$$

$$\frac{C\langle \mathsf{\exists}\langle\tau\rangle\ t\rangle_p}{C\langle \mathsf{T}\rangle \vee t\ u \approx \mathsf{\bot}}\ \text{GExistsHoist}$$

1. p is $\succeq$-eligible in C and $t \in \mathcal{T}^\star_{\mathrm{GH}}$.

The inference rule for argument congruence on the GH level looks as follows:

$$\frac{C \vee t \approx s}{C \vee t\ \bar{u} \approx s\ \bar{u}}\ \text{GArgCong}$$

1. $t \approx s$ is strictly $\succeq$-eligible in $C \vee t \approx s$;;
2. $\bar{u}$ is a tuple of appropriatly typed terms of $\mathcal{T}^\star_{\mathrm{GH}}$..

The rules GExt and GChoice are premise-free inference rules and their conclusions are all the instances obtained by applying the grounding function $\mathcal{G}$ to the axioms (Ext) and (Choice), respectively.

The rules GForallRw and GExistsRw replace ForallRw and ExistsRw, respectively, and use the witness function *GHWit*:

$$\frac{C\langle \forall\langle\tau\rangle\ t\rangle_p}{C\langle t\ \mathit{GHWit}(C,p)\rangle}\ \text{GForallRw}$$

$$\frac{C\langle \exists\langle\tau\rangle\ t\rangle_p}{C\langle t\ \mathit{GHWit}(C,p)\rangle}\ \text{GExistsRw}$$

1. p is $\succeq$-eligible in C

2a. for GForallRw: $\mathcal{F}(C\langle\top\rangle)$ is not a tautology

2b. for GExistsRw: $\mathcal{F}(C\langle\bot\rangle)$ is not a tautology.

The inference system *GFInf* is, like *GHInf*, parameterized by a triple q, where

$$q = (\mathit{GFLitSel}, \mathit{GFBoolSel}, \mathit{GFWit}).$$

Via the bijection $\mathcal{F}$, the parameter triple q of *GHInf* can be translated to a parameter triple $\mathcal{F}(q)$ for *GFInf*. The inference system $\mathit{GFInf}^{\mathcal{F}(q)}$ is obtained by mapping every inference rule using the encoding $\mathcal{F}$, except the rules GArgCong, GExt, and GChoice. Because $\mathit{GFInf}^{\mathcal{F}(q)}$ and GHInf^{q} are connected via the bijection $\mathcal{F}$, they are isomorphic for all rules except GArgCong, GExt, and GChoice. Note that $\mathit{GFInf}^{\mathcal{F}(q)}$ identical to the inference system defined in Section 4.1.3.

The functions $\mathcal{F}$ and $\mathcal{G}$ are extended to inferences:

Definition 23 (Extension of $\mathcal{F}$ and $\mathcal{G}$ to inferences)
Let $q \in \mathcal{Q}$ be a parameter triple and let $\iota \in \mathit{GHInf}^q$ be an inference that stems not from the rules GArgCong, GExt, or GChoice. Then, $\mathcal{F}^q(\iota) \in \mathit{GFInf}^{\mathcal{F}(q)}$ denotes the inference obtained by $\mathit{prems}(\mathcal{F}(\iota)) = \mathcal{F}(\mathit{prems}(\iota))$ and $\mathit{concl}(\mathcal{F}(\iota)) = \mathcal{F}(\mathit{concl}(\iota))$. Given an inference $\iota \in \mathit{HInf}$, the set $\mathcal{G}^q(\iota)$ of ground instances of ι w.r.t. to q is defined to be all inferences $\iota' \in \mathit{GHInf}^q$ such that $\mathit{prems}(\iota') = \mathit{prems}(\iota)\theta$ and $\mathit{concl}(\iota') = \mathit{concl}(\iota)\theta$ for an arbitrary grounding substitution θ.

With the above definition the grounding function $\mathcal{G}$ performs the following mappings, where a pair (Inf_1, Inf_2) denotes the fact, that inference rule Inf_1 is mapped to Inf_2 by $\mathcal{G}$:

- (FluidSup, Sup)
- (FluidBoolHoist, BoolHoist)
- (ForallRw, GForallRw)
- (ExistsRw, GExistsRw)
- (ForallHoist, GForallHoist)
- (ExistsHoist, GExistsHoist)
- (ArgCong, GArgCong)
- (Ext, GExt)
- (Choice, GChoice)

Inferences of other rules of *HInf* are mapped to the identical named counterparts in *GHInf*. The rule FluidLoobHoist needs not to be grounded for refutational completeness. Thus, let $\mathcal{G}^q(\iota) = \mathit{undef}$ for inferences ι of the rule FluidLoobHoist. Now, the sets of redundant inferences w.r.t. a given set of clauses can be given.

Definition 24 (Inference Redundancy)

As for clause redundancy, the sets of redundant inferences w.r.t. a given set of clauses is defined for each level, where each level builds up on the one that comes before it:

- Let $GFRed_{\mathrm{I}}^q : \mathcal{P}(\mathcal{C}_{\mathrm{GF}}) \to \mathcal{P}(GFInf^q)$ be the inference redundancy criterion for level GF, where

$$GFRed_{\mathrm{I}}^q(N) = \{\iota \in GFInf^q \mid prems(\iota) \cap GFRed_{\mathrm{C}}(N) \neq \emptyset; \text{ or } N^{\prec mprem(\iota)} \models concl(\iota)\}.$$

- Let $GHRed_{\mathrm{I}}^q : \mathcal{P}(\mathcal{C}_{\mathrm{GH}}) \to \mathcal{P}(GHInf^q)$ be the inference redundancy criterion for level GH, where

$$GHRed_{\mathrm{I}}^q(N) = \{\iota \in GHInf^q \mid \iota \text{ is a GArgCong, GExt, or GChoice inference and } concl(\iota) \in N \cup GHRed_{\mathrm{C}}(N); \text{ or } \iota \text{ is any other inference and } \mathcal{F}^q(\iota) \in GFRed_{\mathrm{I}}^{\mathcal{F}(q)}(\mathcal{F}(N))\}.$$

- Let $HRed_{\mathrm{I}} : \mathcal{P}(\mathcal{C}_{\mathrm{H}}) \to \mathcal{P}(HInf^q)$ be the inference redundancy criterion for level H, where

$$HRed_{\mathrm{I}}(N) = \{\iota \in HInf \mid \iota \text{ is a FluidLoobHoist inference and } \mathcal{G}(concl(\iota)) \in \mathcal{G}(N) \cup GHRed_{\mathrm{C}}(\mathcal{G}(N)); \text{ or } \iota \text{ is any other inference and } \mathcal{G}^q(\iota) \subseteq GHRed_{\mathrm{I}}(\mathcal{G}(N)) \text{ for all } q \in \mathcal{Q}\}.$$

4.2.7 Refutational Completeness

Like the completeness proof outlined in Section 4.1.5, the proof is divided into three steps:

1. First, static refutational completeness is proved for *GFInf*.
2. By transforming the first-order model obtained by the previous step to a higher-order model, the static refutational completeness of *GHInf* is obtained.
3. The result from step 2 is lifted to *HInf* by instantiating the saturation framework described in Chapter 3.

In the following I will give a proof sketch of each step.

Ground First-Order Level

The result of static refutational completeness is already sketched in Section 4.1.5. However, the theorem can be adapted to the context of the higher-order completeness proof, such that it can be used in the subsequent step. In the following, let R^*_M be the term rewriting system induced by the first-order clause set M as defined in [Num+21].

Theorem 3 (Ground first-order static refutational completeness [Ben+23b, Theorem 52])

Let $q \in \mathcal{F}(Q)$ be a first-order parameter-triple and let $N \subseteq \mathcal{C}_{\mathrm{GF}}$ be a clause set saturated w.r.t. GFInf^q and $\mathit{GFRed}^q_{\mathrm{I}}$ such that $\bot \notin N$. Then, the interpretation $(\mathcal{U}, \mathcal{J})$ induced by the interpretable rewrite system $R^*_{N \setminus \mathit{GFRed}_{\mathrm{C}}(N)}$ is a model of N, i.e., the inference system GFInf^q is statically refutationally complete w.r.t. $\models$ and $(\mathit{GFRed}_{\mathrm{I}}, \mathit{GFRed}_{\mathrm{C}})$.

Ground Higher-Order Level

Let $q = (\mathit{GHLitSel}, \mathit{GHBoolSel}, \mathit{GHWit}) \in Q$ be a parameter triple and let $N \subseteq \mathcal{C}_{\mathrm{GH}}$. By definition, all terms of $\mathcal{T}_{\mathrm{GH}}$ are $\mathsf{Q}_{\approx}$-normal, and thus N is $\mathsf{Q}_{\approx}$-normal. To show static refutational completeness of GHInf^q, N is assumed to be saturated w.r.t. GHInf^q and $\mathit{GHRed}^q_{\mathrm{I}}$ and moreover $\bot \in N$. By construction, $\mathcal{F}(N)$ is also saturated w.r.t. $\mathit{GFInf}^{\mathcal{F}(q)}$ and $\mathit{GFRed}^{\mathcal{F}(q)}_{\mathrm{I}}$. By Theorem 3, the interpretation $R = (\mathcal{U}, \mathcal{J})$ obtained from the interpretable rewrite system $R^*_{\mathcal{F}(N) \setminus \mathit{GFRed}_{\mathrm{C}}(\mathcal{F}(N))}$ is a model of $\mathcal{F}(N)$.

Now, the first-order model $(\mathcal{U}, \mathcal{J})$ needs to be transformed into a higher-order model that satisfies N. To this end, a higher-order interpretation $\mathcal{I}^{\mathrm{GH}} = (\mathcal{U}^{\mathrm{GH}}, \mathcal{J}^{\mathrm{GH}}_{\mathsf{ty}}, \mathcal{J}^{\mathrm{GH}}, (L)^{\mathrm{GH}})$ is derived from $R^*_{\mathcal{F}(N) \setminus \mathit{GFRed}_{\mathrm{C}}(\mathcal{F}(N))}$. First, the universe $\mathcal{U}^{\mathrm{GH}}$ needs to be defined. It must be the case that $\mathcal{J}^{\mathrm{GH}}_{\mathsf{ty}}(\to)(\mathcal{D}_1, \mathcal{D}_2)$ is a subset of the function space from $\mathcal{D}_1$ to $\mathcal{D}_2$ for $\mathcal{D}_1, \mathcal{D}_2 \in \mathcal{U}^{\mathrm{GH}}$. But all the first-order universes $\mathcal{U}_\tau$ are equivalence classes of terms

modulo $R^*_{\mathcal{F}(N)\setminus GFRed_{\mathrm{C}}(\mathcal{F}(N))}$. To repair this issue, two families of functions $\mathcal{E}_\tau$ and $\mathcal{D}_\tau$ are defined by mutual recursion. The function $\mathcal{D}_\tau$ defines the higher-order domain for type τ. The type-indexed function $\mathcal{E}_\tau$ takes as input an element of a first-order universe $\mathcal{U}_\tau$ and produces a corresponding element of $\mathcal{D}_\tau$. $\mathcal{D}_\tau$ is defined for each ground type and $\mathcal{U}^{\mathrm{GH}}$ is the set of all domains $\mathcal{D}_\tau$ for ground τ. If τ is a nonfunctional type, let $\mathcal{D}_\tau = \mathcal{U}_\tau$ and $\mathcal{E} : \mathcal{U}_\tau \to \mathcal{D}_\tau$ is the identity. For functional types, the definition is as follows:

$$\mathcal{D}_{\tau\to\upsilon} = \{\varphi : \mathcal{D}_\tau \to \mathcal{D}_\upsilon \mid \exists s : \tau \to \upsilon.\, \forall u : \tau.\, \varphi\,(\mathcal{E}_\tau([\![\mathcal{F}(u)]\!]_R)) = \mathcal{E}\,([\![\mathcal{F}(s\,u)]\!]_R)\}$$
$$\mathcal{E}_{\tau\to\upsilon} : \mathcal{U}_{\tau\to\upsilon} \to \mathcal{D}_{\tau\to\upsilon}$$
$$\mathcal{E}_{\tau\to\upsilon}([\![\mathcal{F}(s)]\!]_R)\,(\mathcal{E}([\![\mathcal{F}(u)]\!]_R)) = \mathcal{E}_\upsilon\,([\![\mathcal{F}(s\,u)]\!]_F)$$

To show well-definedness of $\mathcal{E}_{\tau\to\upsilon}$ and $\mathcal{D}_{\tau\to\upsilon}$ the following proof obligations have to be discharged:

- $\mathcal{E}_{\tau\to\upsilon}$ is bijective.
- Every element of $\mathcal{U}_{\tau\to\upsilon}$ is of the from $[\![\mathcal{F}(s)]\!]_R$ for some $s \in \mathcal{T}_{\mathrm{GH}}$.
- Every element of $\mathcal{D}_\tau$ is of the form $\mathcal{E}_\tau\,([\![\mathcal{F}(u)]\!]_R)$ for some $u \in \mathcal{T}_{\mathrm{GH}}$.
- The definition does not depend on the choice of s and u.
- It holds that $\mathcal{E}_{\tau\to\upsilon}\,([\![\mathcal{F}(s)]\!]_R) \in \mathcal{D}_{\tau\to\upsilon}$ for every $s \in \mathcal{T}_{\mathrm{GH}}$.

In [Ben+23b] this is accomplished with a rather complex proof using the computability path order [BJR15], Kőnig's lemma [Kőn27], and well-founded induction on terms and substitutions using the left-to-right lexicographic order composed of three different notions of size of λ-terms.

With the well-behaved definition of $\mathcal{E}_\tau$ and $\mathcal{D}_\tau$ in place, the higher-order universe is defined as $\mathcal{U}^{\mathrm{GH}} = \{\mathcal{D}_\tau \mid \tau \text{ is ground}\}$. Since $[\mathsf{T}_0]$ is identified with 1 and $[\bot_0]$ is identified with 0, it follows that $\mathcal{D}_o = \{0, 1\} \in \mathcal{U}^{\mathrm{GH}}$ as required in Section 2.5.1. The definition of the type interpretation is completed by $\mathcal{J}^{\mathrm{GH}}_{\mathsf{ty}}(\kappa)(\mathcal{D}_{\bar\tau}) = \mathcal{U}_{\kappa(\bar\tau)}$ for all $\kappa \in \Sigma_{\mathsf{ty}}$. The interpretation function $\mathcal{J}^{\mathrm{GH}}$ is defined as follows. For nonquantifier symbols $\mathsf{f} : \Pi\bar\alpha_m.\,\tau$ let $\mathcal{J}^{\mathrm{GH}}(\mathsf{f}, \mathcal{D}_{\bar\upsilon_m}) = \mathcal{E}(\mathcal{J}(\mathsf{f}_0^{\bar\upsilon_m}))$. For quantifiers the definitions are

$$\mathcal{J}^{\mathrm{GH}}(\forall, \mathcal{D}_\tau)(f) = \min\{f(a) \mid a \in \mathcal{D}_\tau\}$$
$$\mathcal{J}^{\mathrm{GH}}(\exists, \mathcal{D}_\tau)(f) = \max\{f(a) \mid a \in \mathcal{D}_\tau\}$$

where $f \in \mathcal{J}^{\mathrm{GH}}_{\mathsf{ty}}(\to)(\mathcal{D}_\tau, \{0, 1\})$.

Then, it can be shown that these definitions fulfill all requirements (J1) to (J11). Finally, the designation function $\mathcal{L}$ needs to be defined in a way that the interpretation $\mathcal{I}^{\mathrm{GH}}$ is proper. A designation function receives as input a valuation ξ and a λ-expression $\lambda x.\, t$. From these arguments, a grounding substitution θ is chosen that satisfies $\mathcal{D}_{\alpha\theta} = \xi(\alpha)$

and $\mathcal{E}([\![\mathcal{F}(y\theta)]\!]_R) = \xi(y)$ for all type variables α and all term variables y in $\lambda x.\, t$. The first equation can be fulfilled because in the definition of $\mathcal{I}^{\mathrm{GH}}$ there is a none-to-one correspondence between ground types and domains. The second equation can be obtained by choosing a ground first-order term s such that $[\![s]\!]_R^{\xi} = \mathcal{E}^{-1}(\xi(y))$ and defining $y\theta = \mathcal{F}^{-1}(s)$. The designation function is then defined as $\mathcal{L}^{\mathrm{GH}}(\xi, (\lambda x.\, t)) = \mathcal{E}\left([\![\mathcal{F}(((\lambda x.\, t)\downarrow_{\mathsf{Q}_{\approx}})\theta)]\!]_R\right)$.

Subsequently, it is shown that $\mathcal{I}^{\mathrm{GH}}$ is proper and is moreover a model of N. Because it was assumed that N is saturated and also $\bot \notin N$, the static refutational completeness follows.

Theorem 4 (Ground static completeness [Ben+23b, Theorem 62])
GHInf^q is statically refutationally complete w.r.t. $\models$ and $(GHRed_{\mathrm{I}}^q, GHRed_{\mathrm{C}})$ for every parameter triple $q \in Q$.

Nonground Higher-Order Level

In the last step of the completeness proof, the static refutational completeness of *GHInf* is lifted using the saturation framework and the dynamic refutational completeness of *HInf* can be obtained. To make this possible, several proof obligations have to be taken care of:

- The relation $\models$ is a consequence relation, i.e., conditions (C1) to (C4) must be shown.
- The pair $(GHRed_{\mathrm{I}}^q, GHRed_{\mathrm{C}})$ is a valid redundancy criterion, i.e., properties (R1) to (R4) must hold.
- The function $\mathcal{G}^q$ is a valid grounding function for every $q \in Q$, i.e., requirements (G1) to (G3) must be fulfilled.

Theorem 50 of [Wal+22] can be instantiated for the current context in order to lift the completeness result.

Theorem 5 (Lifting theorem [Ben+23b, Theorem 66])
Assume that *GHInf*^{q} is statically refutationally complete w.r.t. $(GHRed_{\mathrm{I}}^q, GHRed_{\mathrm{C}})$ for every parameter triple $q \in Q$. If for every $N \subseteq \mathcal{C}_{\mathrm{H}}$ that is saturated w.r.t. *HInf* and $HRed_{\mathrm{I}}$ there exists a $q \in Q$ such that $GHInf^q(\mathcal{G}(N)) \subseteq \mathcal{G}^q(HInf(N)) \cup GHRed_{\mathrm{I}}^q(\mathcal{G}(N))$, then also *HInf* is statically refutationally complete w.r.t. $(HRed_{\mathrm{I}}, HRed_{\mathrm{C}})$ and $\models_{\mathcal{G}}$.

In the following, let $N \subseteq \mathcal{C}_{\mathrm{H}}$ be a set of clauses that is saturated w.r.t. *HInf*. In order to use the above theorem, a $q \in Q$ needs to be constructed with the property that every inference $\iota \in GHInf^q$ with $prems(\iota) \in \mathcal{G}(N)$ is either liftable or redundant. More formally,

ι is *liftable* if ι is a $\mathcal{G}^q$-ground instance of an *HInf*-inference from N, and ι is *redundant* if $\iota \in \mathit{GHRed}_\mathrm{I}^q(\mathcal{G}(N))$. The parameter triple $q = (\mathit{GHLitSel}, \mathit{GHBoolSel}, \mathit{GHWit}) \in Q$ is defined in the following way. Every ground clause $C \in \mathcal{G}(N)$ must have at least one corresponding clause $D \in N$ such that $C = D\theta$ for some grounding substitution θ. For every C such a clause D is chosen, which is denoted by $\mathcal{G}^{-1}(C)$. The functions *GHLitSel* and *GHBoolSel* are then chosen such that the selections in C correspond to the selections in $\mathcal{G}^{-1}(C)$ according to Definition 20.

It remains to define the witness function *GHWit*. To this end, let $C \in \mathcal{C}_\mathrm{GH}$ and let p be a green position of a quantifier-headed term $C|_p = \mathsf{Q}\langle\tau\rangle\, t$, let $D = \mathcal{G}^{-1}(C)$ and let θ be a grounding substitution, where $D\theta = C$. Denote with p' the green position corresponding to p in D. If there is no such position p', $\mathit{GHWit}(C, p)$ is defined as an arbitrary term that fulfills the order requirements. Otherwise, let α and x be fresh variables and θ is extended to θ' by defining $\alpha\theta' = \tau$ and $x\theta' = t$. Then, θ' is a unifier of $\mathsf{Q}\langle\alpha\rangle\, x$ and $D|_{p'}$ and thus there exists an idempotent $\sigma \in \mathrm{CSU}(\mathsf{Q}\langle\alpha\rangle\, x, D|_{p'})$ such that for some ρ and all free variables x in $\mathcal{FV}(D) \cup \{x, \alpha\}$ it holds that $x\sigma\rho = x\theta'$. If $\mathsf{Q} = \forall$, let $\mathit{GHWit}(C, p) = \mathsf{sk}_{\Pi\bar{\alpha}.\, \forall\bar{x}.\, \exists z.\, \neg(x\sigma\, z)}\langle\bar{\alpha}\rangle\, \bar{x}\theta$. For the case $\mathsf{Q} = \exists$, let $\mathit{GHWit}(C, p) = \mathsf{sk}_{\Pi\bar{\alpha}.\, \forall\bar{x}.\, \exists z.\, (x\sigma\, z)}\langle\bar{\alpha}\rangle\, \bar{x}\theta$, where $\bar{\alpha}$ are the free type variables and $\bar{x}$ are the free variables occurring in $D|_{p'}$, respectively, in order of first occurrence.

Having defined the parameter triple q, it can be shown that all inferences from $\mathcal{G}(N)$ are either liftable or redundant. The following lemma demonstrates a crucial property of the constructed parameter triple $q = (\mathit{GHLitSel}, \mathit{GHBoolSel}, \mathit{GHWit})$.

Lemma 2 ([Ben+23b, Lemma 67])

Let $C\theta \in \mathcal{C}_\mathrm{GH}$ with $C = \mathcal{G}^{-1}(C\theta)$ and let σ and ρ be substitutions, where $x\sigma\rho = x\theta$ for all $x \in \mathcal{FV}(C)$. If a literal in $C\theta$ is (strictly) $\succeq$-eligible w.r.t. *GHLitSel*, then the corresponding literal in C is (strictly) $\succsim$-eligible w.r.t. σ and *HLitSel*. If a green position in $C\theta$ is $\succeq$-eligible w.r.t. *GHBoolSel* and there exists a corresponding green position in C, then the corresponding position in C is $\succsim$-eligible w.r.t. σ and *HBoolSel*.

I will not retrace in detail the lifting of every inference in the calculus, as this is beyond the scope of this thesis. Nonetheless, I show how the lifting is done by going through the proof that shows how Sup inferences from *GHInf* are either lifted to a Sup or a FluidSup inference on *HInf* or are shown to be redundant. Note that this proof is not given in [Ben+23b] but in [Ben+21] because only the definition of deeply occurring variables has changed, which does not alter the validity of the proofs.

Lemma 3 (Lifting of Sup inferences ([Ben+21, Lemma 52]))
Let $\iota \in \mathit{GHInf}$ be a Sup inference. The inference ι has to look as follows

$$\frac{\overbrace{D'\theta \vee t\theta \approx t'\theta}^{D\theta} \qquad \overbrace{C'\theta \vee s\theta\langle t\theta\rangle_p \mathrel{\dot{\approx}} s'\theta}^{C\theta}}{D'\theta \vee C'\theta \vee s\theta\langle t'\theta\rangle_p \mathrel{\dot{\approx}} s'\theta}\ \textsc{Sup}$$

with $\mathcal{G}^{-1}(D\theta) = D = D' \vee t \approx t' \in N$, $s\theta = s\theta\langle t\theta\rangle_p$, and $\mathcal{G}^{-1}(C\theta) = C = C' \vee s \mathrel{\dot{\approx}} s' \in N$. Denote with p' the longest prefix of p that is a green position of s, which always exists because ε is a green position of every term, and let $u = s|_{p'}$. Then, it holds that ι is liftable if one of the following conditions is satisfied:

(i) u is a deeply occurring variable in C; or

(ii) $p = p'$ and the variable condition holds for D and C; or

(iii) $p \neq p'$ and u is not a variable.

The proof goes as follows, assuming that every term is represented by its η-short β-normal representative. The inference conditions of Sup for ι are that $t\theta \not\preceq t'\theta$, p is $\succeq$-eligible in $C\theta$, $C\theta \not\preceq D\theta$, $t\theta \approx t'\theta$ is strictly $\succeq$-eligible in $D\theta$, $t\theta$ is not a fully applied logical symbol, and if $t'\theta = \bot$, the position p is at the top level of a positive literal.

We have that $u\theta = s\theta|_{p'}$ from lemma 51 of [Ben+21], which shows that $(t|_p)\sigma\downarrow_{\beta\eta} = (t\sigma\downarrow_{\beta\eta})|_p$ for all terms t, substitutions σ and green positions p of both t and $t\sigma\downarrow_{\beta\eta}$. We proceed by a case split on whether $p = p'$, u is not fluid, and u is not a variable deeply occurring in C. In the first case, it will be shown that ι is liftable to a Sup inference of HInf. The second case assumes that either $p \neq p'$, u is fluid, or u is a variable deeply occurring in C and concludes that ι is liftable to a FluidSup inference of HInf.

Case 1: Assume that $p = p'$, u is not fluid, and u is not a variable deeply occurring in C. It follows that $u\theta = s\theta|_{p'} = s\theta|_p = t\theta$. Because θ is a unifier of u and t, there exists some idempotent $\sigma \in \mathrm{CSU}(t, u)$ such that there is some substitution ρ with the property that for every $x \in \mathcal{FV}(D) \cup \mathcal{FV}(C)$ it holds that $x\sigma\rho = x\theta$. The inference ι will be lifted to the following Sup inference $\iota' \in \mathit{HInf}$, which is given by

$$\frac{D' \vee t \approx t' \qquad C' \vee s\langle u\rangle_p \mathrel{\dot{\approx}} s'}{(D' \vee C' \vee s\langle t'\rangle_p \mathrel{\dot{\approx}} s')\sigma}\ \textsc{Sup}$$

It is the case that ι is the $\sigma\rho$-ground instance of ι'. It remains to show that the inference conditions of are liftable, as described next.

- Condition 1: Since u is not fluid, condition 1 is satisfied.
- Condition 2: Also, since u is not a variable deeply occurring in C, condition 2 is satisfied, too.

- Condition 3: In this case, since $p = p'$, condition (iii) cannot hold. Moreover, because of u is not a variable occurring deeply in C, it follows that condition (i) cannot hold either. Thus, it must be the case that the variable condition holds for D and C and hence condition 3 of Sup is satisfied.
- Condition 4: By definition, $\sigma \in \mathrm{CSU}(t, u)$.
- Condition 5: From $t\theta \not\preceq t'\theta$ follows that $t\theta \not\precsim t'\theta$.
- Condition 6: Since position p is $\succeq$-eligible in $C\theta$ it follows that p is $\succsim$-eligible in C w.r.t. σ by Lemma 2.
- Condition 7: From $C\theta \not\preceq D\theta$ follows that $C\theta \not\precsim D\theta$.
- Condition 8: Since $t\theta \approx t'\theta$ is strictly $\succeq$-eligible in $D\theta$ it follows that $t \approx t'$ is strictly $\succsim$-eligible in D w.r.t. θ by Lemma 2.
- Condition 9: Because $t\theta$ is not a fully applied logical symbol it follows that $t\sigma$ is not a fully applied logical symbol, either.
- Condition 10: Assume that $t'\sigma = \bot$. This implies that $t'\sigma\rho = t'\theta = \bot$, which in turn gives that p is at the top level of a positive literal.

Case 2: Assume that (a) $p \neq p'$, (b) u is fluid, or (c) u is a variable deeply occurring in C. First, it is shown that (a) implies (b) or (c). To this end, suppose that (a) is true but neither (b) nor (c) holds. Thus, condition (iii) must hold, that is, u is not variable. Additionally, because (b) does not hold, u cannot be of the form $y\,\bar{u}_n$ for some variable y and $n \geq 1$. If $u = \mathsf{f}\langle\bar{\tau}\rangle\, s_1 \ldots s_n$ with $n \geq 1$, then $u\theta = \mathsf{f}\langle\bar{\tau}\theta\rangle\, s_1\theta \ldots s_n\theta$. It cannot be the case that $n = 0$, since this would imply that $p = p'$, contradicting (a). Hence, $n \geq 1$ and there must be some $1 \leq i \leq n$ such that $p'.i$ is a prefix of p and $s|_{p'.i}$ is a green subterm of s, which contradicts the definition of p'. Thus, u must be a λ-expression. But because $t\theta$ is a proper green subterm of $u\theta$, it follows that $u\theta$ is also not a λ-expression, which leads to the desired contradiction. Hence, we can assume that (b) or (c) holds.

Let $p = p'.p''$, let z be a fresh term variable, and let $\theta' = \theta \uplus \{z \mapsto \lambda y.\, (s\theta|_{p'})\langle y\rangle_{p''}\}$. Then, $(z\,t)\theta' = (s\theta|_{p'})\langle t\theta\rangle_{p''} = s\theta|_{p'} = u\theta = u\theta'$. Because θ' is a unifier of u and $z\,t$, there exists an idempotent $\sigma \in \mathrm{CSU}(z\,t, u)$ such that for some substitution ρ, it holds that $x\sigma\rho = x\theta'$ for $x \in \mathcal{FV}(C) \cup \mathcal{FV}(D) \cup \{z\}$. The inference ι will be lifted to the following FluidSup inference $\iota' \in \mathit{HInf}$, which is given by

$$\frac{D' \vee t \approx t' \qquad C' \vee s\langle u\rangle_{p'} \mathrel{\dot{\approx}} s'}{(D' \vee C' \vee s\langle z\,t'\rangle_{p'} \mathrel{\dot{\approx}} s')\sigma}\ \textsc{FluidSup}$$

It is the case that ι is the $\sigma\rho$-ground instance of ι'. It remains to show that the inference conditions of are liftable, as described next.

- Condition 1: Since either (b) or (c) holds, the condition is satisfied.

- Condition 2: It holds that z is a fresh variable.
- Condition 3: By construction, $\sigma \in \mathrm{CSU}(z\,t, u)$.
- Condition 4: Assume that $(z\,t)\sigma = (z\,t')\sigma$. It follows that $(z\,t)\sigma\rho = (z\,t')\sigma\rho$, which implies that $(z\,t)\theta' = (z\,t')\theta'$. But since $t\theta' = t\theta \neq t'\theta = t'\theta'$, we have that $(z\,t)\theta' \neq (z\,t')\theta'$, obtaining a contradiction. Thus, $(z\,t)\sigma \neq (z\,t')\sigma$.
- Conditions 5 to 10: As in case 1.

This concludes the proof of Lemma 3. It can then be shown that every Sup inference that is not captured by this lemma is redundant. This fact is given in the following lemma.

Lemma 4 ([Ben+21, Lemma 53])
Let $\iota \in \mathit{GHInf}$ be a Sup inference such that $\mathit{prems}(\iota) \subseteq \mathcal{G}(N)$ which is not captured by Lemma 3. Then ι is redundant, i.e., $\iota \in \mathit{GHRed}^q_{\mathrm{I}}(\mathcal{G}(N))$.

At this point, every inference of *GHInf* is either liftable or redundant. Thus, Theorem 5 can be applied to obtain static refutational completeness of *HInf* w.r.t. $\models_{\mathcal{G}}$ and $(\mathit{HRed}_{\mathrm{I}}, \mathit{HRed}_{\mathrm{C}})$. Using the saturation framework [Wal+22], the dynamic refutational completeness of *HInf* w.r.t.$\models_{\mathcal{G}}$ and $(\mathit{HRed}_{\mathrm{I}}, \mathit{HRed}_{\mathrm{C}})$ is derived.

Since, the goal is to have refutational completeness w.r.t. Tarski entailment $\models$ instead of Herbrand entailment $\models_{\mathcal{G}}$, it can be shown that $N \models \bot$ if and only if $N \models_{\mathcal{G}} \bot$ for every $\mathsf{Q}_{\approx}$-normal set of clauses $N \subseteq \mathcal{C}_{\mathrm{H}}$. Thus, dynamic refutational completeness for *HInf* w.r.t. $\models$ and $(\mathit{HRed}_{\mathrm{I}}, \mathit{HRed}_{\mathrm{C}})$ is obtained, with the condition that the initial clause set must be $\mathsf{Q}_{\approx}$-normal. Moreover, this result can be used to show dynamic refutational completeness also for the Skolem-aware entailment $\models\!\!\approx$, provided that the initial clause set does not contain any Skolem symbols.

4.2.8 Saturation Procedure

In this section, the fundamental mechanism of Zipperposition's saturation procedure is given, as described in [Vuk+21]. As in the first-order case, it is a given clause procedure but due to the complications regarding higher-order unification it is more involved. That is, an inference rule no longer has a single conclusion but a potentially infinite number of conclusions if the unification problem yields an infinite stream of incomparable unifiers. Thus, it is not possible to exhaustively enumerate these unifiers for a single inference rule because one can get stuck without getting a refutation because another inference rule needed to be applied. Some provers, for example Leo-III and Vampire 4.4, choose to enumerate only a subset of all possible conclusions of inferences. This leads to incompleteness and also the right size of the subset is very hard to choose. If the size is too big, the prover may spend too much time trying to solve unification constraints and if the size is too small, the unifiers may not be found that are needed for a successful proof.

To circumvent this issue, Zipperposition uses a modification of a given clause procedure, where the computation of unifiers and the application of inferences are interleaved. This modification alone does not solve the problem described above. The enumeration of elements of CSUs needs to be *fair*, in the sense that the prover tries to compute new elements for all given CSUs and does not get stuck indefinitely at a single unification problem. In Zipperposition, this is achieved by representing the set $\mathrm{CSU}(t, u)$ as a lazily computed stream [Oka99, Section 4.2]. Each element of the stream is either the empty set $\emptyset$ or a set consisting of a single unifier of t and u. This design is used because the unification procedure needs to return the empty set after a given amount of computation steps if no new unifier has been found.

Algorithm 2 (Higher-order given clause procedure)

function ExtractClause(Q, *stream*)
  maybe_clause ← pop and compute first element of *stream*
  if *stream* is not empty **then**
    add *stream* to Q with an increased weight
  return *maybe_clause*

function HeuristicProbe(Q)
  $(\mathit{collected\_clauses}, i) \leftarrow (\emptyset, 0)$
  while $i < K_{\text{best}}$ and Q is not empty **do**
    $(\mathit{maybe\_clause}, j) \leftarrow (\emptyset, 0)$
    while $J < K_{\text{retry}}$,Q is not empty, and *maybe_clause* $= \emptyset$ **do**
      stream ← pop the lowest weight stream in Q
      maybe_clause ← ExtractClause(Q, *stream*)
      $j \leftarrow j + 1$
    collected_clauses ← *collected_clauses* ∪ *maybe_clause*
    $i \leftarrow i + 1$
  return *collected_clauses*

function FairProbe(Q, *num_oldest*)
  collected_clauses ← $\emptyset$
  oldest_streams ← pop *num_oldest* oldest streams from Q
  for *stream* in *oldest_streams* **do**
    collected_clauses ← *collected_clauses* ∪ ExtractClause(Q, *stream*)
  return *collected_clauses*

```
function ForceProbe(Q)
  collected_clauses ← ∅
  while Q is not empty and collected_clauses = ∅ do
    collected_clauses ← FairProbe(Q, |Q|)
  if Q and collected_clauses are empty then status ← Satisfiable
  else status ← Unknown
  return (status, collected_clauses)

function GivenClause(P)
  A ← ∅
  Q ← empty priority queue
  status ← Unknown
  i ← 0
  while status = Unknown do
    if P is not empty then
      given ← pop chosen clause from P and simplify it using A
      if given = ⊥ then status ← Unsatisfiable
      else
        A ← A ∪ {given}
        for stream in streams of inferences between given and clauses in A do
          if stream is not empty then P ← P ∪ ExtractClause(Q, stream)
        i ← i + 1
        if i mod K_fair = 0 then P ← P ∪ FairProbe(Q, ⌊i/K_fair⌋)
        else P ← P ∪ HeuristicProbe(Q)
    else // P is empty
      (status, forced_clauses) ← ForceProbe(Q)
      P ← P ∪ forcedClauses
  return status
```

There is much information to unpack in Algorithm 2. First of all, as in Algorithm 1, the variables P and A contain the set of passive and active clauses, respectively. The given clause procedure is invoked by calling GivenClause, where parameter P is the initial clause set. This function operates as follows. First, the active set A is initialized to be empty and Q is an empty priority queue that will contain streams of clauses with some corresponding weight. In a loop, a clause is chosen from P and simplified. If it is the empty clause, the status gets set to Unsatisfiable, which is subsequently returned as the result. Otherwise, the given clause is added to A. Then, for every stream which is a result of an inference between given and a clause of A, the result of ExtractClause(Q, *stream*) is added to P. The function ExtractClause computes the first element of *stream*, adds the stream to Q and returns the computed element of the stream. Note that this element may either be the empty set or a singleton set $\{C\}$ containing a clause C, which is a result of the inference that *stream* represents.

Back in GivenClause, the loop counter is incremented and then checked if FairProbe

should be called. How often FairProbe is called is determined by the parameter K_{fair}, which is 70 by default. In FairProbe, the *num_oldest* streams are removed from Q and for every such stream ExtractClause is called. This is essential to achieve fairness, because otherwise the oldest streams may never be accessed. If FairProbe is not called in the current iteration, the result of HeuristicProbe is added to P. This function tries to extract up to K_{best} clauses from the streams that are most promising w.r.t. to the assigned weight. If the first element of the most promising stream is $\emptyset$, the stream is inserted again into Q and the next stream is chosen. This procedure is iterated until either K_{retry} times the $\emptyset$ has been encountered or a stream produced a new clause.

If P is empty, the function ForceProbe is called to eagerly search for a new clause. If no new clauses were found and Q is the empty queue, the status can be set to Satisfiable. Otherwise, if Q is not empty or new clauses could be produced, the status Unknown is returned along with the newly produced clauses.

If the employed unification procedure coincides with standard first-order unification on first-order terms, this given clause procedure behaves as Algorithm 1 when invoked on a first-order problem.

CHAPTER 5

Constraint Superposition Calculus

One weakness of the presented superposition calculus for higher-order logic is the need to enumerate elements of CSUs. In particular, the unification of terms that are variable-headed, called *flex-flex pairs* due to the flexibility in structure permitted by the variable heads, are a source of explosiveness w.r.t. the search space. Even the unification problem $\{x\ y \stackrel{?}{=} z\ \mathsf{a}\}$, where x, y, and z are variables and a is a constant symbol, has infinitely many unifiers that are pairwise incomparable. Due to restrictions regarding ordering and fairness, it is necessary to also enumerate solutions to such unification problems.

But it is not always necessary to compute all solutions to such problems. Consider a unification problem $\{s_1 \stackrel{?}{=} t_1, \ldots, s_i \stackrel{?}{=} t_i\}$ where every term s_i and t_i is of the form $\lambda\bar{x}.\, y\,\bar{u}$ such that y is a free variable. For every base type υ, let z_υ be a fresh variable of type υ that does not occur in the unification problem. A solution to this unification problem is a substitution θ that maps every variable $y : \bar{\tau}_n \to \upsilon$, where υ is a base type, occurring as a head, to a term of the form $\lambda\bar{x}_n.\, z_\upsilon$. It is easy to see that this is indeed a unifier of every pair $s_i \stackrel{?}{=} t_i$. Using η-long form, let $s_i = \lambda\bar{x}_m.\, y\,\bar{u}_n$ and let $t_i = \lambda\bar{x}_m.\, z\,\bar{r}_k$. Because of the use of η-long, the number of leading λ-binders has to match. Then, applying θ yields the following result:

$$
\begin{aligned}
(\overbrace{\lambda\bar{x}_m.\, y\,\bar{u}_n}^{s_i})\theta &\stackrel{?}{=} (\overbrace{\lambda\bar{x}_m.\, z\,\bar{r}_k}^{t_i})\theta \\
\lambda\bar{x}_m.\ \underbrace{y\theta}_{\text{apply definition of } \theta}\ \bar{u}_n\theta &\stackrel{?}{=} \lambda\bar{x}_m.\ \underbrace{z\theta}_{\text{apply definition of } \theta}\ \bar{r}_k\theta \\
\lambda\bar{x}_m.\ \underbrace{(\lambda\bar{x}_n.\, z_\upsilon)\ \bar{u}_n\theta}_{\beta\text{-reduce}} &\stackrel{?}{=} \lambda\bar{x}_m.\ \underbrace{(\lambda\bar{x}_k.\, z_\upsilon)\ \bar{r}_k\theta}_{\beta\text{-reduce}} \\
\lambda\bar{x}_m.\, z_\upsilon &\stackrel{?}{=} \lambda\bar{x}_m.\, z_\upsilon
\end{aligned}
$$

Since both terms s_i and t_i were arbitrary, it follows that θ is a solution to the unification problem. Huet developed an algorithm that solves unification problems until only flex-flex

pairs remain [Hue75], which is called *Huet's preunification algorithm*. He then used this approach for a constrained resolution calculus for higher-order logic [Hue72], which is refutationally complete w.r.t. Henkin semantics if infinitely many extensionality axioms are given [Ben02, Theorem 9]. In this calculus higher-order unification can be postponed such that when deriving the empty clause with a set of unification constraints it is tested if the constraints permit a solution. Because of this the enumeration of unifiers for flex-flex pairs is no longer necessary. The goal of this work is to use this approach for the presented superposition calculus for higher-order logic, which gives rise to the name *constraint superposition calculus*.

This chapter is based on unpublished work of Alexander Bentkamp, Jasmin Blanchette, Uwe Waldmann, and myself. It is structured as follows. The first two chapters discuss Huet's preunification algorithm and its optimized variant, which is implemented in Zipperposition. Another notable change is a switch to the *locally nameless representation* for λ-terms [Cha11], which is discussed in Section 5.3. The following section defines parameters of the constraint superposition calculus such as term order and selection functions. Afterwards, the inference rules are given and the redundancy criterion is defined. In Section 5.7 a proof sketch of refutational completeness is given. The last two sections describe the employed saturation algorithm and the corresponding implementation in Zipperposition.

5.1 Huet's Preunification Procedure

Before the preunification of Huet can be discussed, the needed terminology needs to be introduced. As already described in Section 4.2.1, a unification problem, also called system, is a multiset of unordered term pairs of the same type, written as $\{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$. The pair $x \stackrel{?}{=} t$ is said to be in *solved form* w.r.t. unification problem $\{x \stackrel{?}{=} t\} \cup E$ if x is variable and $x \notin \mathcal{FV}(E) \cup \mathcal{FV}(t)$. A system is in solved form if all its pairs are in solved form. Such a system always looks has the form $\{x_1 \stackrel{?}{=} t_1, \ldots, x_n \stackrel{?}{=} t_n\}$, where $x_1, \ldots, x_n$ are pairwise distinct variables and $\{x_1, \ldots, x_n\} \cap \bigcup_{i=1}^{n} \mathcal{FV}(t_i) = \emptyset$. It holds that a system which is in solved form is always a set rather than a multiset because every free variable only occurs once on one side of a unification constraint. Moreover, it is trivial to construct a substitution θ that is a solution of a solved system E. To see this, let $E = \{x_1 \stackrel{?}{=} t_1, \ldots, x_n \stackrel{?}{=} t_n\}$ be system in solved form. Then, a substitution is induced by E, denoted by θ_E, where $\theta_E = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is an idempotent most-general unifier of E [SG89, Lemma 3.4].

Huet's preunification algorithm operates on η-long λ-terms that are in so-called *head normal form*. A λ-term t is in head normal form if it is of the form $\lambda\bar{x}.\, a\, \bar{t}$, where a is either a free variable, a bound variable or a constant. If this is the case, a is called the *head* of t. If a is free variable, it is called a *flex* head and otherwise it is called a *rigid* head. A term is called flex (rigid) if its head is flex (rigid). A unification constraint $s \stackrel{?}{=} t$ is either called rigid-rigid, flex-rigid, or flex-flex, depending on the heads of s and t. Moreover, the preunification algorithm assumes that λ-terms in head normal form are

also given in their η-long form. This has the effect that for every two terms $\lambda\bar{x}_n.\, t$ and $\lambda\bar{x}_m.\, s$ that have the same type it follows that $n = m$. If E is in pre-solved form, let θ_E denote the substitution equal to $\theta_{E'}$, where E' is the restriction of E to unification constraints in solved form.

A unification constraint $s \stackrel{?}{=} t$ is in *pre-solved form* w.r.t. unification problem E if it is in solved form w.r.t. E or s and t are both flex terms. A system is in pre-solved form if every unification constraint is in pre-solved form. A substitution θ is a *preunifier* of s and t if $s\theta$ and $t\theta$ only differ at subterms that are variable-headed [Pre98]. A *complete set of preunifiers* on a set X of variables for two terms s and t is a set U of preunifiers of s and t, such that for every preunifier θ of s and t, there exists some $\rho \in U$ and some σ such that $x\rho\sigma$ and $x\theta$ only differ at subterms that are variable-headed for every $x \in X$.

When solving flex-rigid constraints $\lambda\bar{x}.\, y\ \bar{t} \stackrel{?}{=} \lambda\bar{x}.\, a\ \bar{s}$, where y is a flex head and a is a rigid head, it must hold that if θ (pre-)solves the pair, then $(\lambda\bar{x}.\, y\ \bar{t})\theta$ must be headed by a as well, since a substitution does not change a rigid head. There are two possibilities to choose, when searching for an instantiation for y:

- If a is some constant f, then y is replaced by a term that is headed by f. This is called *imitation*.
- Otherwise, an argument of y is used as a replacement for y, which is called *projection*.

The detailed notions are introduced next.

Definition 25 (Imitation Binding [VBN20])

Let y be a free variable with type $\bar{\tau}_n \to \upsilon$, where υ is not functional, and let f be a constant of type $\bar{\gamma}_n \to \upsilon$, where $n, m \geq 0$. The *imitation binding* of f for y, denoted by $\mathcal{IB}(\mathsf{f}, y)$, is then defined as

$$\{y \mapsto \lambda\bar{x}_n.\, \mathsf{f}\ (y_1\ \bar{x}_n) \ldots (y_m\ \bar{x}_n)\}$$

where $\bar{y}_m$ are fresh free variables.

The involved types are as follows:

- Every bound variable x_i is of type τ_i.
- Every fresh free variable y_i is of type $\bar{\tau}_n \to \gamma_i$.

Definition 26 (Projection Binding [VBN20])
Let y be a free variable with type $\bar{\tau}_n \to \upsilon$, where υ is not functional, $n > 0$, and let i be such that $1 \leq i \leq n$ and $\tau_i = \bar{\gamma}_m \to \upsilon$. The i^{th} *projection binding* of y, denoted by $\mathcal{PB}(i, y)$, is then defined as

$$\{y \mapsto \lambda \bar{x}_n.\, x_i\, (y_1\, \bar{x}_n) \ldots (y_m\, \bar{x}_n)\}$$

where $\bar{y}_m$ are fresh free variables.

If τ_i is not of the form $\bar{\gamma}_m \to \upsilon$, then $\mathcal{PB}(i, y)$ is not defined. The involved types are as in Definition 25.

Originally, Huet developed his preunification algorithm using two procedures called Simpl and Match [Hue75]. However, a more modern representation of this algorithm and also other unification procedures is to use a set of rules that define transitions [SG89; VBN20]. Thus, Algorithm 3 is formalized as a set of transitions. Given two terms s and t that need to be unified, the procedure starts with the unification problem $\{s \stackrel{?}{=} t\}$ and builds a tree, where $\{s \stackrel{?}{=} t\}$ is the root. A node is a leaf node if it is $\bot$ or it is a system in solved form. Otherwise, all possible transitions are applied to the node, which produce its child nodes. With this approach various kinds of tree search algorithms can be used to enumerate preunifiers. For transitions, where the resulting system is of the form $\{y \stackrel{?}{=} t\} \uplus E$, where $y \notin \mathcal{FV}(t) \cup \mathcal{FV}(E)$, the variable y is not written in η-long form in order to emphasize that this pair is in solved form and should not be considered again for transitions. This is the only exception, where terms are not represented in η-long form.

Transitions may be applied in any order, but a good heuristic for an implementation is to first check if Fail is applicable to cut the search tree as early as possible. Afterwards, rules Delete, Decompose, and Bind should be applied as often as possible before the explosive rules Imitate and Project are considered.

Algorithm 3 (Huet's preunification algorithm)

Fail $\{\lambda\bar{x}.\, a\ \bar{s} \stackrel{?}{=} \lambda\bar{x}.\, b\ \bar{t}\} \uplus E \longrightarrow \bot$
where a and b are different rigid heads

Delete $\{t \stackrel{?}{=} t\} \uplus E \longrightarrow E$

Bind $\{\lambda\bar{x}.\, y\ \bar{x} \stackrel{?}{=} t\} \uplus E \longrightarrow \{y \stackrel{?}{=} t\} \uplus E\theta$
where y is a flex head, $y \notin \mathcal{FV}(t)$, and $\theta = \{y \mapsto t\}$

Decompose $\{\lambda\bar{x}.\, a\ \bar{s}_m \stackrel{?}{=} \lambda\bar{x}.\, a\ \bar{t}_m\} \uplus E \longrightarrow \{\lambda\bar{x}_n.\, s_i \stackrel{?}{=} \lambda\bar{x}_n.\, t_i \mid 1 \leq i \leq m\} \uplus E$
where a is a rigid head

Imitate $\{\lambda\bar{x}.\, y\ \bar{s} \stackrel{?}{=} \lambda\bar{x}.\, a\ \bar{t}\} \uplus E \longrightarrow \{y \stackrel{?}{=} y\theta\} \uplus E\theta$
where a is some constant g and θ is an imitation of g for y, i.e., $\theta = \mathcal{IB}(\mathsf{g}, y)$

Project $\{\lambda\bar{x}_n.\, y\ \bar{s} \stackrel{?}{=} \lambda\bar{x}_n.\, a\ \bar{t}\} \uplus E \longrightarrow \{y \stackrel{?}{=} y\theta\} \uplus E\theta$
where a is a rigid head, $1 \leq i \leq n$, and θ is a projection for y w.r.t. i, that is, $\theta = \mathcal{PB}(i, y)$

Huet's preunification algorithm was proven to be sound and complete. In the following, $\longrightarrow^*$ denotes the reflexive-transitive closure of the transition relation $\longrightarrow$ defined in Algorithm 3.

Theorem 6 (Soundness of Huet's preunification algorithm [SG89, Theorem 5.6])

Let E and E' be unification problems, such that $E \longrightarrow^* E'$ and E' is in pre-solved form, then $\theta_{E'}|_{\mathcal{FV}(E)}$ is a preunifier of E.

Theorem 7 (Completeness of Huet's preunification algorithm [SG89, Theorem 5.7])

If σ is a preunifier of unification problem E, there exists a sequence of transitions $E \longrightarrow^* E'$, where E' is in pre-solved form and $\theta_{E'}|_{\mathcal{FV}(E)}$ is more general than σ.

Thus, it is possible to employ Huet's preunification algorithm to enumerate the set of preunifiers of two terms.

5.2 Optimized Preunification Procedure

There are several possible optimizations that can be applied to Algorithm 3, that are also presented in [VBN20]. First of all, the algorithm is changed to operate on raw λ-terms. That is, λ-terms without implicit α-equivalence. Moreover, it is no longer assumed that terms are in head normal form, as this will be only lazily enforced. Additionally, terms

are also not assumed to be in η-long form. This is only needed for the outermost term and it would be unnecessary to always give all subterms in η-long form. This property is also going to be lazily enforced. To this end, let $t{\downarrow_{\mathsf{h}}}$ denote the term that is obtained from t by repeated β-reduction of the leftmost outermost redex until the term is in head normal form. Note that this can also be achieved by computing $t{\downarrow_{\beta}}$, but it is simply not needed to β-reduce the whole term. Rules that enforce the naming of bound variables, outermost η-expansion, and reduction to head normal form are $\mathsf{Normalize}_{\alpha\eta}$ and $\mathsf{Normalize}_{\beta}$.

A further improvement is that a state is not represented by a single unification problem E, but a rather a pair (E, θ) consisting of a unification problem E and a substitution θ that represents the current partial solution at this state. This has the effect that in rules such as Bind, Imitate, or Project a substitution does not need to be applied to the whole unification problem, but rather composed with the substitution of the current node. Because of this, a new rule Dereference is introduced that applies the substitution to a flex head to ensure that the head agrees with the corresponding substitution.

Last, but not least, oracles are supported. An oracle is a procedure that enumerates all elements of a CSU of a decidable unification fragment. Such decidable fragments are for example first-order, pattern [Nip93], functions-as-constructors [LM21], and fixpoint [Hue75] unification.

The improved algorithm is given in Algorithm 4. In contrast to Algorithm 3, transitions may no longer be applied in any order, but more expensive rules are only allowed if other rules are not applicable. This is also used to normalize raw λ-terms, as described above, before other rules may inspect the terms. When enumerating preunifiers for terms s and t, the procedure starts with $(\{s \stackrel{?}{=} t\}, \mathsf{id})$ as the root node, where id denotes the identity substitution. A leaf node is either a substitution, which is a preunifier of s and t, or $\bot$, which denotes failure of the corresponding path. Otherwise, the children of an internal node are computed using the given transitions of the algorithm.

Algorithm 4 (Optimized preunification algorithm)

Normalize$_{\alpha\eta}$ $(\{\lambda\bar{x}_m.\ s \stackrel{?}{=} \lambda\bar{y}_n.\ t\} \uplus E, \theta) \longrightarrow (\{\lambda\bar{x}_m.\ s \stackrel{?}{=} \lambda\bar{x}_m.\ t'\ x_{n+1} \dots x_m\} \uplus E, \theta)$
where $m \geq n$, $\bar{x}_m \neq \bar{y}_n$, and $t' = t\{y_1 \mapsto x_1, \dots, y_n \mapsto x_n\}$

Normalize$_{\beta}$ $(\{\lambda\bar{x}.\ s \stackrel{?}{=} \lambda\bar{x}.\ t\} \uplus E, \theta) \longrightarrow (\{\lambda\bar{x}.\ s{\downarrow_{\mathsf{h}}} \stackrel{?}{=} \lambda\bar{x}.\ t{\downarrow_{\mathsf{h}}}\} \uplus E, \theta)$
where s or t is not in head normal form

Dereference $(\{\lambda\bar{x}.\ y\ \bar{s} \stackrel{?}{=} t\} \uplus E, \theta) \longrightarrow (\{\lambda\bar{x}.\ (y\theta)\ \bar{s} \stackrel{?}{=} t\} \uplus E, \theta)$
where y is a flex head and $y \neq y\theta$

Succeed $(E, \theta) \longrightarrow \theta$
where none of the previous rules are applicable and all unification constraints in E are flex-flex

Fail $(\{\lambda\bar{x}.\ a\ \bar{t} \stackrel{?}{=} \lambda\bar{x}.\ b\ \bar{s}\} \uplus E, \theta) \longrightarrow \bot$
where none of the previous rules are applicable and a and b are different rigid heads

Delete $(\{t \stackrel{?}{=} t\} \uplus E, \theta) \longrightarrow (E, \theta)$
where none of the previous rules are applicable

OracleSuccess $(\{s \stackrel{?}{=} t\} \uplus E, \theta) \longrightarrow (E, \rho\theta)$
where none of the previous rules are applicable, an oracle computed a finite CSU U for the unification problem $\{s\theta \stackrel{?}{=} t\theta\}$, and $\rho \in U$.
If multiple oracles found a CSU, only one is used

OracleFail $(\{s \stackrel{?}{=} t\} \uplus E, \theta) \longrightarrow \bot$
where none of the previous rules are applicable and an oracle decided that there exists no solution for the unification problem $\{s\theta \stackrel{?}{=} t\theta\}$

Decompose $(\{s \stackrel{?}{=} t\} \uplus E, \theta) \longrightarrow (\{\lambda\bar{x}.\ s_i \stackrel{?}{=} \lambda\bar{x}.\ t_i \mid 1 \leq i \leq m\} \uplus E, \theta)$
where none of the previous rules are applicable, $s = \lambda\bar{x}.\ a\ \bar{s}_m$, $t = \lambda\bar{x}.\ a\ \bar{t}_m$, and a is a rigid head

Imitate $(\{\lambda\bar{x}.\ y\ \bar{s} \stackrel{?}{=} \lambda\bar{x}.\ a\ \bar{t}\} \uplus E, \theta) \longrightarrow (\{\lambda\bar{x}.\ y\ \bar{s} \stackrel{?}{=} \lambda\bar{x}.\ a\ \bar{t}\} \uplus E, \rho\theta)$
where a is some constant g and ρ is an imitation of g for y, i.e., $\rho = \mathcal{IB}(\mathsf{g}, y)$, and none of the rules Normalize$_{\alpha\eta}$ to Decompose are applicable

Project $(\{\lambda\bar{x}.\ y\ \bar{s} \stackrel{?}{=} \lambda\bar{x}.\ a\ \bar{t}\} \uplus E, \theta) \longrightarrow (\{\lambda\bar{x}.\ y\ \bar{s} \stackrel{?}{=} \lambda\bar{x}.\ a\ \bar{t}\} \uplus E, \rho\theta)$
where a is a rigid head, $1 \leq i \leq n$, and ρ is a projection for y w.r.t. i, that is, $\rho = \mathcal{PB}(i, y)$ and none of the rules Normalize$_{\alpha\eta}$ to Decompose are applicable

The algorithm is designed such that Imitate and Project can be applied in parallel but only if previous rules cannot be applied, because these rules introduce many fresh variables,

which may lead to an explosion of the search space. Dereference partially normalizes the term w.r.t. to the current substitution, such that the head of the terms in the system are synchronized with the state of the substitution. That is, it should not be the case that the algorithm decides that $y \stackrel{?}{=} \mathsf{g}$ is a flex-rigid pair, if $y\theta$ is actually a constant w.r.t. to the corresponding substitution θ.

Often, oracles have the precondition that their input already is in the fragment that is decided by the oracle. For this algorithm, the oracles have to be able to take any input and discover if the problem is included in the respective fragment or if this is not the case [VBN20].

5.3 Locally Nameless Representation

The constraint superposition calculus uses the locally nameless representation for terms of the simply typed λ-calculus, which is based on *De Bruijn indices* [de 72]. Not only is this representation closer to the actual implementation in Zipperposition, but it also has several other advantages. For example, one gets α-equivalence for free and it allows to redefine green subterms, such that more terms are covered with the new definition. First, the syntax is formalized and then also the semantics have to be slightly adapted.

5.3.1 Syntax

In this representation, a bound variable is represented using a De Bruijn index, which is a natural number that indicates the number of λ-binders that are between the bound variable and its corresponding λ-binder. The locally named representation of terms, as defined in Section 2.5, is also called *nominal*, since bound variables are named. For example, the nominal term $\lambda z.\, \lambda x.\, (y\ z\ (\lambda x.\, x)\ x)$ would be given by $\lambda\, \lambda\, (y\ 1\ (\lambda\, 0)\ 0)$ in locally nameless representation. In this example, the free variable y remains unchanged, the bound occurrence of z is replaced by the De Bruijn index 1 because the λ-binder introducing x is between the λ-binder that introduces z. Moreover, in the innermost λ-expression, the bound occurrence of x is replaced by 0 because the innermost λ-binder is the last occurrence that binds x. The last occurrence of the bound variable x is also replaced by 0, but this De Bruijn index refers to the first λ-binder above the subterm $y\,1\,(\lambda\,0)\,0$. One advantage of the locally nameless notation is that α-equivalence of λ-terms is free, because the naming of bound variables is exactly specified by De Bruijn indices.

Note that terms such as $\lambda\, 1$ are most of the time not described, because the De Bruijn index 1 points to λ-binder that is present in the term. Such occurrences of are called *leaking*. Thus, the sets of λ-preterms, λ-terms, preterms, and terms are introduced. In Section 2.5, also raw λ-terms were defined, but the handling of α-renaming is no longer needed with the locally nameless representation.

Let $(\Sigma_{\mathsf{ty}}, \Sigma)$ be a higher-order type signature, and let $\mathcal{V}$ be an infinite set of term variables. To handle the types of leaking De Bruijn indices, an environment is used to provide the expected types thereof. An environment E is a list of types over Σ_{ty} written as

$[\tau_0, \ldots, \tau_{n-1}]$. The first element of E is τ_0 and has index 0, whereas the last element of E is τ_{n-1} with index $n-1$. The length of E, denoted by $|E|$, is n in this case. When adding typings to the environment, the notation $\upsilon :: E$ is used to denote the environment resulting from adding υ to the front of E. The empty list is written as $[]$. Hence, $[\tau_0, \ldots, \tau_{n-1}]$ could also be written as $\tau_0 :: \tau_1 :: \ldots :: \tau_{n-1} :: []$, where $::$ associates to the right. The notation $E \mathbin{!!} n$ denotes the value at index n of E, where the result is only defined if $n < |E|$, that is

$$(x :: xs) \mathbin{!!} n = \begin{cases} x & \text{if } n = 0 \\ xs \mathbin{!!} (n-1) & \text{otherwise} \end{cases}$$

The set of λ-preterms w.r.t. environment E is inductively defined as follows:

- A variable $x : \tau \in \mathcal{V}$ is a λ-preterm of type τ w.r.t. E.
- If $\mathsf{f} : \Pi\bar{\alpha}_n.\, \tau \in \Sigma$ and $\bar{\upsilon}_n$ is a tuple of types, then $\mathsf{f}\langle\bar{\upsilon}_n\rangle$ is a λ-preterm of type $\tau\{\bar{\alpha}_n \mapsto \bar{\upsilon}_n\}$ w.r.t. E.
- If $n \in \mathbb{N}$, $n < |E|$, and $E \mathbin{!!} n = \tau$, then the De Bruijn index $n\langle\tau\rangle$ is a λ-preterm of type τ w.r.t. E.
- If $t : \tau$ is a λ-preterm w.r.t. $\upsilon :: E$, then $\lambda\langle\upsilon\rangle\, t$ is a λ-preterm of type $\upsilon \to \tau$ w.r.t. E.
- If $s : \upsilon \to \tau$ and $t : \upsilon$ are both λ-preterms w.r.t. E, then $s\, t$ is a λ-preterm of type τ w.r.t. E.

If E is the empty list $[]$, it may be omitted. Moreover, if t is a λ-preterm w.r.t. $[]$, then t is a λ-term, since it contains no leaking De Bruijn indices. Notions like substitution, subterms, β-reduction, and η-reduction are defined for the locally nameless notation as expected. The set of *(pre)terms* consists of the $\beta\eta$-equivalence classes of λ-(pre)terms, and all notions are lifted to (pre)terms. As a convention, most of the time the β-normal η-short representative is used when dealing with (pre)terms. For clarity, a nominal representation may be used, as in sections 5.1 and 5.2.

Note that the subterm of a λ-term is not always a λ-term. To see why this is the case, consider $\lambda\langle\tau\rangle\, 0$, where the subterm 0 is not a λ-preterm w.r.t. $[]$, since the empty environment provides no type for De Bruijn index 0. When using nominal λ-terms, the subterm x of $\lambda x.\, x$ may be recognized as a free variable when given without context.

When working with λ-expressions $\lambda\, t$, it is sometimes needed to replace the De Bruijn index bound by the outermost λ binder by some variable x to produce a term t'. This has the effect that the resulting term t' is still a valid term, while t by itself may not be a term, because the removal of the λ binder could lead to a leaking De Bruijn index. This operation is called *variable opening* and is defined as follows. Given a λ-expression $\lambda\, t$ and a variable x, the variable opening of $\lambda\, t$ w.r.t. x, denoted by t^x, is given by

$t^x = \mathtt{open}(t, 0, x)$. The function $\mathtt{open}$ proceeds by recursion on the structure of t as follows [Cha11]:

$$\mathtt{open}(t,k,x) = \begin{cases} x & \text{if } t \text{ is a De Bruijn index equal to } k \\ n & \text{if } t \text{ is a De Bruijn index not equal to } k \\ y & \text{if } t \text{ is a free variable } y \\ \mathtt{open}(t_1,k,x)\ \mathtt{open}(t_2,k,x) & \text{if } t \text{ is an application } t_1\ t_2 \\ \mathtt{open}(t',k+1,x) & \text{if } t \text{ is of the form } \lambda\, t' \end{cases}$$

If $\lambda\, t$ is a valid term, then also t^x is a valid term. The variable x does not need to be fresh, but it is often required for the concrete application of variable opening.

5.3.2 Semantics

The semantics of the simply typed λ-calculus in locally nameless representation has to be slightly adapted. Let $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}, \mathcal{L})$ be an interpretation and let ξ be a valuation. The λ-designation function $\mathcal{L}$ is still a mapping from valuations and terms that are λ-expressions of type τ to elements of $[\![\tau]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}$. The denotation of λ-terms is changed to $[\![\lambda\, t]\!]^{\xi}_{\mathcal{I}} = \mathcal{L}(\xi, \lambda\, t)$. The interpretation $\mathcal{I}$ is *proper* if $[\![\lambda\, t]\!]^{\xi}_{\mathcal{I}}(a) = [\![t^x]\!]^{\xi[x\mapsto a]}_{\mathcal{I}}$ for all terms that are λ-expressions and all valuations ξ, where x is a fresh variable. Note that interpretations are only defined for terms and not for the set of all preterms.

5.4 Term Orders, Selection Functions, and Eligibility

Like the calculus presented in Section 4.2, the constraint superposition calculus is parameterized by a strict and a nonstrict term order, and a literal and Boolean subterm selection function. Their definitions have to be slightly altered, as specified next.

An order $\succ$ is a strict ground term order if it is a well-founded strict total order on ground terms and it satisfies conditions (O1) to (O3) of Definition 11, such that the adapted definition of green subterms is used. Since all quantifiers are preprocessed, condition (O4) is not required. Strict and nonstrict term orders are defined as in Definition 12, with the extension that for a strict term order $\succ$ the following property is required:

(O4') Let t and s be terms such that $t \succ s$ and let θ be a substitution such that all variables in $t\theta$ and $s\theta$ are nonfunctional. Then it must be the case that if $s\theta$ contains a nonfunctional variable, then $t\theta$ must also contain the variable.

A literal selection function is a mapping from a clause C to a subset of its negative literals, where the literals in the subset are called *selected* in C. A Boolean subterm selection function is a function which maps a clause to a subset of its green positions with Boolean subterms. These positions are called *selected* in C. The following restrictions must be obeyed:

- A subterm must not be selected if it is $\top$, $\bot$, or it is a variable-headed term.
- A subterm must not be selected if it is at the top-level position on either side of a positive literal.

Eligibility is used as in Definition 14.

5.5 Inference Rules

Since unification can be postponed and done in tandem with performing inferences, the enumeration of CSUs is no longer required. However, when doing an inference it may be useful to try to find either a most general unifier, try to compute the first elements of a CSU, or perform preunification for a fixed amount of time before storing the unification constraint as is. One could, for example, check if the unification problem is efficiently solvable by some oracle. To this end, the notion of *complete sets of substitutions* is used.

Definition 27 (Complete sets of substitutions)

A complete set of substitutions on a set X of variables for constraints T, denoted by $\mathrm{CSS}_X(T)$, is a set S of substitutions with the following property: For every unifier θ of T there exists some $\rho \in S$ and a substitution σ such that $x\rho\sigma = x\theta$ for every $x \in X$. In the following, $\mathrm{CSS}_X(T)$ denotes an arbitrary complete set of substitutions for T and it is assumed that every contained substitution is idempotent on X. As for CSUs, it is required that every substitution of $\mathrm{CSS}_X(T)$ unifies the types of every unification constraint contained in T. Usually, X is the set of free variables of the involved clauses and will be left implicit.

Intuitively, every element of a complete set of substitutions is either a solution of the given constraints or can be extended to be a solution of them. Examples for complete sets of substitutions are complete sets of unifiers, complete sets of preunifiers, and even the singleton set containing the identity substitution.

The formal definition of clauses with unification constraints is given next.

Definition 28 (Constrained clauses)

A *constraint* is a finite set of unordered pairs of terms, written as $[\![t_1 \stackrel{?}{=} s_1, \ldots, t_n \stackrel{?}{=} s_n]\!]$. A *constrained clause* is a pair consisting of a clause C and a constraint T, denoted by $C[\![T]\!]$. Let $\mathcal{C}_\mathrm{H}$ denote the set of all constrained clauses. If T is empty, we may write C instead of $C[\![T]\!]$. The semantics for constrained clauses is as expected, i.e., $C[\![T]\!]$ is satisfied by an interpretation if at least one literal of C is satisfied and for every constraint $t \stackrel{?}{=} s$ of T it holds that t and s are syntactically equal.

We define $(C[\![t_1 \stackrel{?}{=} s_1, \ldots, t_n \stackrel{?}{=} s_n]\!])\theta = (C\theta)[\![t_1\theta \stackrel{?}{=} s_1\theta, \ldots, t_n\theta \stackrel{?}{=} s_n\theta]\!]$ for all substitutions θ, clauses C and terms $t_1, \ldots t_n, s_1, \ldots s_n$.

The definition of green subterms is changed such that green subterms are a subset of *orange subterms*.

Definition 29 (Orange subterms and orange positions)
An *orange position* is a tuple of natural numbers that indicates an orange subterm as follows:

- The empty position ε is an orange position of every term t. The orange subterm at ε is t.
- If p is an orange position of s_i, then $i.p$ is an orange position of the term $a\ \bar{s}$, where a is either a constant symbol or a De Bruijn index. The orange subterm at $i.p$ is the orange subterm of s_i at p.
- If p is an orange position of t, then $1.p$ is an orange position of the term $\lambda\, t$. The orange subterm at $1.p$ is the orange subterm of u at p.

Using the notion of orange subterms, a green subterm is an orange subterm that does not contain leaking De Bruijn indices. If u is a green subterm of t at position p, we write $t\langle u\rangle_p$, where p may be omitted. For example, consider the term $\mathsf{f}\ (x\ \mathsf{a}\ z)\ (\lambda\ \mathsf{g}\ \mathsf{b}\ (\mathsf{h}\ 0))$. Its orange subterms are the term itself, $x\ \mathsf{a}\ z$, $\lambda\ \mathsf{g}\ \mathsf{b}\ (\mathsf{h}\ 0)$, $\mathsf{g}\ \mathsf{b}\ (\mathsf{h}\ 0)$, b, $(\mathsf{h}\ 0)$, and 0. On the other hand, its green subterms are the term itself, $x\ \mathsf{a}\ z$, and $\lambda\ \mathsf{g}\ \mathsf{b}\ (\mathsf{h}\ 0)$, and b. Note that the only green subterm under the λ-abstraction is the subterm b because all other orange subterms contain the De Bruijn index 0. Moreover, the subterm b is not green w.r.t. to the old definition of green terms as in Section 4.2.

Let $\succ$ be a strict term order and let $\succsim$ be a nonstrict term order. We also globally fix a literal selection function *HLitSel* and a Boolean subterm selection function *HBoolSel*. In the following, the rules of the constraint superposition calculus are presented.

$$\frac{\overbrace{D' \vee t \approx t'}^{D}\ [\![T]\!] \qquad C\langle u\rangle\ [\![S]\!]}{(D' \vee C\langle t'\rangle\ [\![T, S, t \stackrel{?}{=} u]\!])\sigma}\ \textsc{Sup}$$

1. $\sigma \in \mathrm{CSS}(T, S, t \stackrel{?}{=} u)$;
2. u is not variable-headed;
3. $t\sigma \not\precsim t'\sigma$;
4. the position of u is $\succsim$-eligible in C w.r.t. σ;
5. $C\sigma \not\precsim D\sigma$;
6. $t \approx t'$ is strictly $\succsim$-eligible in D w.r.t. σ;
7. $t\sigma$ is not a fully applied logical symbol;
8. if $t'\sigma = \bot$, the position of u is at the top level of a positive literal.

$$\frac{\overbrace{D' \vee t \approx t'}^{D} \llbracket T \rrbracket \qquad C\langle u \rangle \llbracket S \rrbracket}{(D' \vee C\langle z\, t' \rangle \llbracket T, S, z\, t \stackrel{?}{=} u \rrbracket)\sigma}\ \textsc{FluidSup}$$

1. $\sigma \in \mathrm{CSS}(T, S, z\, t \stackrel{?}{=} u)$;
2. u is headed by a functional variable;
3. - 8. conditions 3 to 8 from Sup;
9. z is a fresh variable;
10. $(z\, t')\sigma \neq (z\, t)\sigma$.

$$\frac{\overbrace{C' \vee u \not\approx u'}^{C} \llbracket S \rrbracket}{(C' \llbracket S, u \stackrel{?}{=} u' \rrbracket)\sigma}\ \textsc{ERes}$$

1. $\sigma \in \mathrm{CSS}(S, u \stackrel{?}{=} u')$;
2. $u \not\approx u'$ is $\succsim$-eligible in C w.r.t. σ.

$$\frac{\overbrace{C' \vee u \approx v' \vee u \approx v}^{C} \llbracket S \rrbracket}{(C' \vee v \not\approx v' \vee u \approx v' \llbracket S, u \stackrel{?}{=} u' \rrbracket)\sigma}\ \textsc{EFact}$$

1. $\sigma \in \mathrm{CSS}(S, u \stackrel{?}{=} u')$;
2. $u\sigma \not\preceq v\sigma$;
3. $u \approx v$ is $\succsim$-eligible in C w.r.t. σ.

$$\frac{\overbrace{C' \vee s \approx s'}^{C} \llbracket S \rrbracket}{(C' \vee s\, x \approx s'\, x \llbracket S \rrbracket)\sigma}\ \textsc{ArgCong}$$

1. σ is the most general type substitution such that $s\sigma$ is functional (that is, $\sigma = \mathsf{id}$ if s is functional or $\{\alpha \mapsto (\beta \to \gamma)\}$ for fresh β and γ if s is of type α for some type variable α);
2. $s \approx s'$ is strictly $\succsim$-eligible in C w.r.t. σ;
3. x is a fresh variable.

$$\frac{C\langle u\rangle[\![S]\!]}{(C\langle\bot\rangle \vee u \approx \top[\![S]\!])\sigma}\ \textsc{BoolHoist}$$

1. σ is the most general type substitution such that $u\sigma$ is of Boolean type (that is, $\sigma = \mathsf{id}$ if u is of Boolean type or $\{\alpha \mapsto o\}$ if u is of type α for some type variable α);
2. u is neither variable-headed nor a fully applied logical symbol;
3. the position of u is $\succsim$-eligible in C w.r.t. σ;
4. the occurrence of u is not at the top level of a positive literal.

$$\frac{\overbrace{C' \vee s \approx s'}^{C}\ [\![S]\!]}{(C'[\![S, s \stackrel{?}{=} \bot, s' \stackrel{?}{=} \top]\!])\sigma}\ \textsc{FalseElim}$$

1. $\sigma \in \mathrm{CSS}(S, s \stackrel{?}{=} \bot, s' \stackrel{?}{=} \top)$;
2. $s \approx s'$ is strictly $\succsim$-eligible in C w.r.t. σ.

$$\frac{C\langle u\rangle[\![S]\!]}{(C\langle\bot\rangle \vee x \approx y[\![S, u \stackrel{?}{=} x \mathrel{\approx\!\!\!\approx} y]\!])\sigma}\ \textsc{EqHoist}$$

$$\frac{C\langle u\rangle[\![S]\!]}{(C\langle\top\rangle \vee x \approx y[\![S, u \stackrel{?}{=} x \mathrel{\not\approx\!\!\!\approx} y]\!])\sigma}\ \textsc{NeqHoist}$$

1. $\sigma \in \mathrm{CSS}(S, u \stackrel{?}{=} t)$, where t is
   - $x \mathrel{\approx\!\!\!\approx} y$ for EqHoist;
   - $x \mathrel{\not\approx\!\!\!\approx} y$ for NeqHoist.
2. x and y are fresh variables;
3. the position of u is $\succsim$-eligible in C w.r.t. σ;
4. if the head of u is a variable, it must be applied and the affected literal must be of the form $u \approx \top$, $u \approx \bot$, or $u \approx v$ for a variable-headed term v.

$$\frac{C\langle u\rangle[\![S]\!]}{(C\langle t'\rangle[\![S, t \stackrel{?}{=} u]\!])\sigma}\ \textsc{BoolRw}$$

1. $\sigma \in \mathrm{CSS}(S, t \stackrel{?}{=} u)$ and (t, t') is on of the following pairs, where x is a fresh variable:

$(\neg\bot, \top)$	$(\bot \wedge \top, \bot)$	$(\bot \vee \top, \top)$	$(\bot \to \top, \top)$
$(\neg\top, \bot)$	$(\top \wedge \top, \top)$	$(\top \vee \top, \top)$	$(\top \to \top, \top)$
$(\bot \wedge \bot, \bot)$	$(\bot \vee \bot, \bot)$	$(\bot \to \bot, \top)$	$(x \approx x, \top)$
$(\top \wedge \bot, \bot)$	$(\top \vee \bot, \top)$	$(\top \to \bot, \bot)$	$(x \not\approx x, \bot)$

2. the position of u is $\gtrsim$-eligible in C w.r.t. σ;
3. if the head of u is a variable, it must be applied and the affected literal must be of the form $u \approx \top$, $u \approx \bot$, or $u \approx v$ for a variable-headed term v.

$$\frac{C\langle u\rangle[\![S]\!]}{(C\langle z\ \bot\rangle \vee x \approx \top[\![S, z\ x \stackrel{?}{=} u]\!])\sigma}\ \textsc{FluidBoolHoist}$$

1. u is headed by a functional variable;
2. z and x are fresh variables;
3. $\sigma \in \mathrm{CSS}(S, z\ x \stackrel{?}{=} u)$;
4. $(z\ \bot)\sigma \neq (z\ x)\sigma$;
5. $x\sigma \notin \{\top, \bot\}$;
6. the position of u is $\gtrsim$-eligible in C w.r.t. σ.

$$\frac{C\langle u\rangle[\![S]\!]}{(C\langle z\ \top\rangle \vee x \approx \bot[\![S, z\ x \stackrel{?}{=} u]\!])\sigma}\ \textsc{FluidLoobHoist}$$

1. u is headed by a functional variable;
2. z and x are fresh variables;
3. $\sigma \in \mathrm{CSS}(S, z\ x \stackrel{?}{=} u)$;
4. $(z\ \top)\sigma \neq (z\ x)\sigma$;
5. $x\sigma \notin \{\top, \bot\}$;
6. the position of u is $\gtrsim$-eligible in C w.r.t. σ.

Finally, axiom Ext is used for extensionality reasoning.

$$z\ (\mathsf{diff}\langle\alpha, \beta\rangle\ z\ y) \not\approx y\ (\mathsf{diff}\langle\alpha, \beta\rangle\ z\ y) \vee z \approx y \qquad \textsc{(Ext)}$$

5.6 Redundancy Criterion

Given a higher-order signature $(\Sigma_{\mathsf{ty}}, \Sigma)$ a first-order signature $(\Sigma_{\mathsf{ty}}, \Sigma_{\mathrm{F}})$ is constructed as described in Section 4.2.6. Additionally, De Bruijn indices will also be encoded in the first-order level. To this end, for each monomorphic type $\bar{\tau} \to \tau$ with $m \geq 0$ and for every $n \in \mathbb{N}$, a first-order symbol $\mathsf{db}_j^{n,\bar{\tau}\to\tau}$ is introduced with argument type $\tau_1 \times \cdots \times \tau_j$ and result type $\tau_{j+1} \to \cdots \to \tau_m \to \tau$ for every $0 \leq j \leq m$. Here, the superscript n will denote the actual value of the index. Note that on the first-order level $\to$ is an uninterpreted type constructor. Moreover, for all monomorphic types τ and υ a first-order symbol lam^{τ} with argument type υ and result type $\tau \to \upsilon$. This constant is used to encode λ-expressions $\lambda\langle\tau\rangle\ t$ of type $\tau \to \upsilon$ as $\mathsf{lam}^{\tau}(t')$, where t' is the encoded version of t.

When translating higher-order constrained clauses to first-order clauses, the notion of *ground closures*, which are a subset of constrained clauses, is used.

Definition 30 (Ground closures)

A *ground closure* is a constrained clause $C[\![T]\!]$, where T is of the form $x_1 \stackrel{?}{=} t_1, \ldots, x_n \stackrel{?}{=} t_n$, all variables x_i are pairwise different and include all variables occurring in C, and t_i are ground terms. Ground closures are denoted by $C \cdot \theta$, where $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Let $\mathcal{C}_{\mathrm{G}}$ denote the set of all ground closures. A closure $C \cdot \theta$ is true in an interpretation if $C\theta$ is.

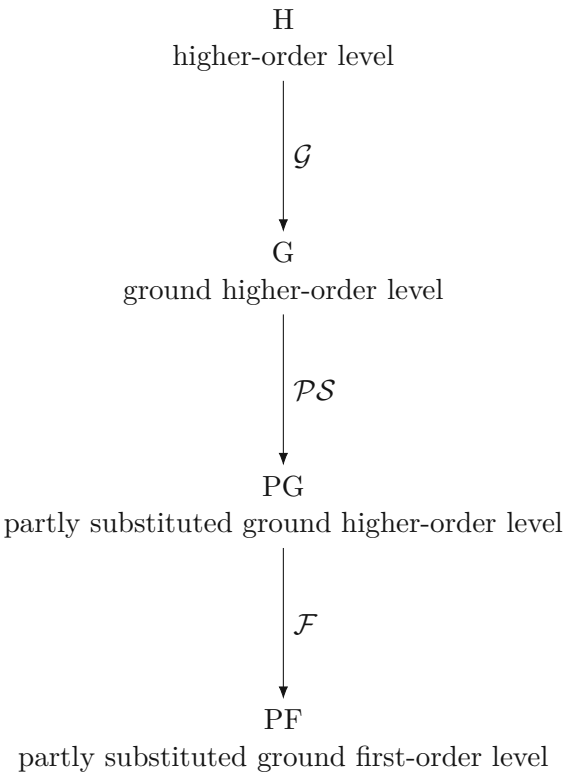
Note that for ground closures $C \cdot \theta$ the clause C can contain variables but every variable occurring in C has to be mapped to a ground term by θ. Ground closures will be used by the grounding function $\mathcal{G}$, because we only want to consider ground instances of a constrained clause $C[\![T]\!]$ that satisfy T.

The redundancy criterion is based on a translation to ground monomorphic first-order logic with an interpreted Boolean type. That is, the logic used in Section 4.1 restricted to the variable- and quantifier-free fragment. Moreover, as in Section 4.2.6, the sketch of the completeness proof as the redundancy criterion are developed using levels. But rather than three levels, four levels are used, with corresponding sets of (constrained) clauses or closures, terms, and inference system, which are given below:

- Level H: The higher-order level H deals with higher-order terms as defined in Section 5.3 and constrained clauses as defined in Definition 28. Let $\mathcal{T}_{\mathrm{H}}$ and $\mathcal{C}_{\mathrm{H}}$ denote the set of terms and constrained clauses on this level, respectively. Its inference system is called *HInf*, whose rules are given in Section 5.5.
- Level G: The ground higher-order level G contains ground higher-order terms and ground closures as defined below in Definition 31. Let $\mathcal{T}_{\mathrm{G}}$ denote the set of ground higher-order terms and $\mathcal{C}_{\mathrm{G}}$ denote the set of all ground closures. The inference system called *GInf*.

- Level PG: This level is called the partly substituted ground higher-order level and is a fragment of G, where only nonfunctional variables occur. Let $\mathcal{T}_{\mathrm{PG}}$ denote the set of ground higher-order terms without functional variables and let $\mathcal{C}_{\mathrm{PG}}$ denote the set of all ground closures without functional variables. Its inference system is denoted by *PGInf*.
- Level PF: The partly substituted ground first-order level contains firt-order closures and first-order terms with interpreted Booleans. Let $\mathcal{T}_{\mathrm{PF}}$ denote the set of first order terms over the signature $(\Sigma_{\mathsf{ty}}, \Sigma_{\mathrm{F}})$ and let $\mathcal{C}_{\mathrm{PF}}$ denote the set of ground first-order closures over $\mathcal{T}_{\mathrm{PF}}$. Moreover, let *PFInf* denote the inference system for level PF.

Levels H, G, and PG use higher-order logic as defined in Section 2.5 with the new concepts of constrained clauses and closures. The level PF uses first-order logic with interpreted Booleans as given in Section 2.2. The inference systems are discussed in Section 5.7. The levels are connected via a grounding function $\mathcal{G}$, a partial substitution mapping PS and a first-order encoding function $\mathcal{F}$ as visualized below:



The definitions of these functions will be discussed next. The transition from level H to level G is done via the grounding function $\mathcal{G}$.

Definition 31 (Grounding function $\mathcal{G}$)

The grounding function $\mathcal{G} : \mathcal{C}_{\mathrm{H}} \to \mathcal{P}(\mathcal{C}_{\mathrm{G}})$ maps a constrained clause $C[\![T]\!] \in \mathcal{C}_{\mathrm{H}}$ to a set of ground closures of the form $C \cdot \theta \in \mathcal{C}_{\mathrm{G}}$, such that $T\sigma\theta$ is true. The closure $C \cdot \theta$ is called a *ground instance* of $C[\![T]\!]$.

Note that if a list of constraints T is unsatisfiable, for example, if T contains a constraint that has different rigid heads, then $\mathcal{G}(C[\![T]\!]) = \emptyset$. The next step is the function $\mathcal{PS}$, which takes a ground closure $\cdot\theta \in \mathcal{C}_{\mathrm{G}}$ and maps it to a ground closure $C' \cdot \theta' \in \mathcal{C}_{\mathrm{PG}}$ that contains only nonfunctional variables. This process is called *partial substitution*.

Definition 32 (Partial substitution function $\mathcal{PS}$)

Given a ground closure $C \cdot \theta \in \mathcal{C}_{\mathrm{G}}$, the partial substitution function $\mathcal{PS} : \mathcal{C}_{\mathrm{G}} \to \mathcal{C}_{\mathrm{PG}}$ is defined in two steps. Two substitutions $\mathfrak{p}(\theta)$ and $\mathfrak{q}(\theta)$ are constructed as follows. First, $\mathfrak{p}(\theta)$ consists of all type variable mappings in θ. Moreover, for each mapping $y \mapsto t$ in θ with $y\theta \neq y$, define $y\mathfrak{p}(\theta)$ to be the term resulting from replacing each nonfunctional green subterm at a green position p in t of type τ by a fresh variable $y_p : \tau$. If a nonfunctional green subterm is contained inside another nonfunctional green subterm, then only the outermost nonfunctional green subterm is replaced by such a fresh variable. The substitution $\mathfrak{q}(\theta)$ is given by $y_p\mathfrak{q}(\theta) = y\theta|_p$ for every fresh variable y_p introduced by $\mathfrak{p}(\theta)$. Finally, let $\mathcal{PS}(C \cdot \theta) = C\mathfrak{p}(\theta) \cdot \mathfrak{q}(\theta)$.

The complicated definition of $\mathcal{PS}$ deserves to be accompanied by an illustrating example. To this end, consider $(x\,\mathsf{g} \vee y) \cdot \theta$ with $\theta = \{\alpha \mapsto (\iota \to o), x \mapsto \lambda\,\mathsf{f}\,(0\,(\mathsf{h}\,\mathsf{b})), y \mapsto \mathsf{c}\}$, where $x : \alpha$, $y : o$, $b : \iota$, $\mathsf{c} : o$, $\mathsf{h}, \mathsf{g} : \iota \to \iota$, and $\mathsf{f} : \iota \to o$. Then:

$$\mathfrak{p}(\theta) = \{\alpha \mapsto (\iota \to o), x \mapsto \lambda\,\mathsf{f}\,(0\,x_{1.1.1}), y \mapsto y_\varepsilon\}$$
$$\mathfrak{q}(\theta) = \{x_{1.1.1} \mapsto \mathsf{h}\,\mathsf{b}, y_\varepsilon \mapsto \mathsf{c}\}$$

And finally:

$$\begin{aligned}\mathcal{PS}((x\,\mathsf{g} \vee y) \cdot \theta) &= (x\,\mathsf{g} \vee y)\mathfrak{p}(\theta) \cdot \mathfrak{q}(\theta)\\ &= ((\lambda\,\mathsf{f}\,(0\,x_{1.1.1}))\,\mathsf{g} \vee y_\varepsilon) \cdot \mathfrak{q}(\theta)\\ &= (\mathsf{f}\,(\mathsf{g}\,x_{1.1.1}) \vee y_\varepsilon) \cdot \mathfrak{q}(\theta)\end{aligned}$$

Note that $\mathfrak{p}(\theta)$ and $\mathfrak{q}(\theta)$ are defined such that if θ grounds a term t, then $t\mathfrak{p}(\theta)$ contains only nonfunctional variables and $t\theta = t\mathfrak{p}(\theta)\mathfrak{q}(\theta)$. That is, variables can only occur at the top level for Boolean variables or as arguments of De Bruijn indices or function symbols but never occur applied to arguments. These variables can then be used by the first-order encoding function $\mathcal{F}$, where they are mapped to first-order variables. Otherwise, functional variables would have no first-order counterpart and could not be translated, which means that the refutational completeness cannot be lifted. In the last step, $\mathcal{F}$ is used to go from level PG to level PF.

Definition 33 (First-order encoding function $\mathcal{F}$)

Let $\mathcal{F} : \mathcal{T}_{\mathrm{PG}} \to \mathcal{T}_{\mathrm{PF}}$ be defined by structural recursion on its argument:

$$\begin{aligned}
\mathcal{F}(x) &= x \\
\mathcal{F}(\lambda\langle\tau\rangle\ t) &= \mathsf{lam}^{\tau}(\mathcal{F}(t)) \\
\mathcal{F}(\mathsf{f}\langle\bar{\tau}\rangle\ t_1\ \ldots\ t_n) &= \mathsf{f}_n^{\bar{\tau}}(\mathcal{F}(t_1), \ldots, \mathcal{F}(t_n)) \\
\mathcal{F}(m\langle\tau\rangle\ t_1\ \ldots\ t_n) &= \mathsf{db}_n^{m,\tau}(\mathcal{F}(t_1), \ldots, \mathcal{F}(t_n))
\end{aligned}$$

$\mathcal{F}$ is lifted to closures $C \cdot \theta \in \mathcal{T}_{\mathrm{PG}}$ by defining $\mathcal{F}(C \cdot \theta) = \mathcal{F}(C) \cdot \mathcal{F}(\theta)$, where $\mathcal{F}(C)$ maps each side of each literal individually and $\mathcal{F}(\theta) = \mathcal{F}(\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}) = \{x_1 \mapsto \mathcal{F}(t_1), \ldots, x_n \mapsto \mathcal{F}(t_n)\}$.

Following other completeness proofs for superposition calculi with constraints [Bac+92; NR92], only a subset of ground closures are used for showing refutational completeness, which are called *order-irreducible* ground instances. This notion is defined for first-order logic as follows.

Definition 34 (Order-irreducibility)

A ground closure literal $L \cdot \theta$ is *order-irreducible* w.r.t. a ground term rewrite system R if for all variables x in L, $x\theta$ is irreducible w.r.t. every rule $s \longrightarrow t \in R$, where $L\theta \succ s \approx t$. A ground closure $C \cdot \theta \in \mathcal{C}_{\mathrm{PF}}$ is order-irreducible w.r.t. R if all its literal are order-irreducible w.r.t. R. Given $N \subseteq \mathcal{C}_{\mathrm{PF}}$, the set $\mathrm{irred}_R(N)$ denotes all order-irreducible ground closures in N w.r.t. R.

To ensure that there are enough order-irreducible ground instances of constrained clauses $D[\![U]\!]$ whenever a ground closure $C \cdot \theta$ is order-irreducible, the following definition is introduced.

Definition 35 (Trust)

A ground closure $C \cdot \theta \in \mathcal{C}_{\mathrm{G}}$ *trusts* a ground instance $D \cdot \rho$ of $D[\![U]\!] \in \mathcal{C}_{\mathrm{H}}$ if for each variable x in D,

(Trust1) x does not appear in U; or

(Trust2) for every literal $L \in D$ containing x, there exists a literal $K \in C$ and a substitution σ such that $z\theta = z\sigma\rho$ for all variables z in C, and $L \preceq K\sigma$.

The next lemma will be helpful when using the notion of trust for the justification of simplification rules.

Lemma 5 (Unconstrained trust)

Let $C \cdot \theta \in \mathcal{C}_{\mathrm{G}}$ and $D[\![U]\!] \in \mathcal{C}_{\mathrm{H}}$. If U is empty, i.e., $D[\![U]\!] = D$, then $C \cdot \theta$ trusts every ground instance $D \cdot \rho$ of D.

The proof just uses condition Trust1 for every variable x in D, which holds trivially since U is empty and thus x does not appear therein.

With these definitions in place, a simple version of clause redundancy, fittingly named *simple clause redundancy*, is introduced. This version of redundancy makes it easier to justify simplification rules, because the general form of redundancy criteria for constrained superposition calculi are very hard to apply. This notion of redundancy is based on work of Nieuwenhuis and Rubio [NR92].

Definition 36 (Simple clause redundancy)

Let $N \subseteq \mathcal{C}_{\mathrm{H}}$ and $C[\![T]\!] \in \mathcal{C}_{\mathrm{H}}$. The constrained clause $C[\![T]\!]$ is called *simply redundant* w.r.t. N, denoted by $C[\![T]\!] \in \mathit{HRed}^{\star}_{\mathrm{C}}(N)$, if for every $C \cdot \theta \in \mathcal{G}(C[\![T]\!])$ one of the following conditions holds:

(SCR1) There exist ground instances $D_i \cdot \rho_i$ of clauses $D_i[\![U_i]\!] \in N$, where $1 \leq i \leq n$, such that

- (SCR1a) $\mathcal{F}(\mathcal{P}(\{D_1 \cdot \rho_1, \ldots, D_n \cdot \rho_n\})) \models \mathcal{F}(\mathcal{P}(C \cdot \theta))$; and
- (SCR1b) for all $1 \leq i \leq m$ it holds that $D_i \cdot \rho_i \prec C \cdot \theta$; and
- (SCR1c) for all $1 \leq i \leq m$ it holds that $C \cdot \theta$ trusts the instance $D_i \cdot \rho_i$ of $D_i[\![U_i]\!]$.

(SCR2) There exists a ground instance $D \cdot \rho$ of some $D[\![U]\!] \in N$ such that

- (SCR2a) $D\rho = C\theta$; and
- (SCR2b) $C[\![T]\!] \sqsupset_{C\theta} D[\![U]\!]$; and
- (SCR2c) $C \cdot \theta$ trusts the instance $D \cdot \rho$ of $D[\![U]\!]$.

Condition SCR1 is a generalization of standard superposition redundancy to higher-order constrained clauses, while the second condition SCR2 can be used for subsumption. It is parameterized by a family of well-founded partial orders $\{\sqsupset_C |\ C \in \mathcal{C}_{\mathrm{G}}\}$ on $\mathcal{C}_{\mathrm{H}}$. If this family is chosen correctly, partial unification of constraints can be made a simplification rule. To this end, let $D[\![U]\!] \in \mathcal{C}_{\mathrm{H}}$ and let $n_{D[\![U]\!]}$ be the smallest number of unification steps w.r.t. the specific algorithm used needed to find a substitution σ such that there is a ρ, where $C = D\sigma\rho$ and $\sigma\rho$ is a solution of U. If there is no such σ, then $n_{D[\![U]\!]} = \infty$. Then, let $D[\![U]\!] \sqsupset_C D'[\![U']\!]$ iff $n_{D[\![U]\!]} > n_{D'[\![U']\!]}$. This definition has the effect that partial unification makes clauses smaller w.r.t. $\sqsupset_C$.

In an implementation, the following simplification rule Unif can be used to perform partial unification of constraints.

$$\frac{C[\![T]\!]}{C[\![U_1]\!] \quad \cdots \quad C[\![U_n]\!]}\ \textsc{Unif}$$

1. for all ground substitutions θ, $T\theta$ is true iff there is a i, such that $U_i\theta$ is true;
2. $C[\![T]\!] \sqsupset_{C\theta} C[\![U_i]\!]$ for all i and all grounding substitutions θ, where $T\theta$ is true.

Now, it can be proven that the notion of simple clause redundancy is able to justify Unif. To this end, it must be shown that $C[\![T]\!]$ is simply redundant w.r.t. $\{C[\![U_i]\!] \mid 1 \leq i \leq n\}$, i.e., $C[\![T]\!] \in \mathit{HRed}^{\star}_{\mathrm{C}}(\{C[\![U_i]\!] \mid 1 \leq i \leq n\})$.

For a proof, let $C \cdot \theta$ be an arbitrary ground instance of $C[\![T]\!]$. Because of condition 1 of Unif, there exists an i, where $U_i\theta$ is true. To show simple redundancy condition SCR2 is employed, where $C \cdot \theta$ is used for $D \cdot \rho$ and $C[\![U_i]\!]$ for $D[\![U]\!]$, which is possible since $C \cdot \theta$ is a ground instance of $C[\![U_i]\!]$. Clearly, it holds that $C\theta = C\theta$, which proves condition SCR2a. Moreover, condition SCR2b follows from condition 2 of Unif. Finally, to discharge condition SCR2c, condition Trust2 is used, where σ is the identity substitution. □

Related to partial unification, the following Weaken simplification rule allows to substitute a solved pair which occurs in the unification constraints of a clause.

$$\frac{C[\![y \equiv u, T]\!]}{(C[\![T]\!])\{y \mapsto u\}}\ \textsc{Weaken}$$

1. y is a variable which does not occur in u;
2. $C[\![y \equiv u, T]\!] \sqsupset_{C\theta} (C[\![T]\!])\{y \mapsto u\}$ for all grounding substitutions θ, where $(y \equiv u, T)\theta$ is true.

To show that also Weaken can be justified by simple clause redundancy, the proof obligation $C[\![y \equiv u, T]\!] \in \mathit{HRed}^{\star}_{\mathrm{C}}(\{(C[\![T]\!])\{y \mapsto u\}\})$ has to be discharged. Let $C \cdot \theta$ be a ground instance of $C[\![y \equiv u, T]\!]$. It must be the case that $y\theta = u\theta$ because θ is a solution for $y \equiv u$. Condition SCR2 is applied, where $(C[\![T]\!])\{y \mapsto u\}$ is used for $D[\![U]\!]$ and $C\{y \mapsto u\} \cdot \theta$ is used for $D \cdot \rho$. Condition SCR2a holds, i.e., $C\theta = C\{y \mapsto u\}\theta$ because $y\theta = u\theta$. Condition SCR2b follows from condition 2 of Weaken. Finally, for condition SCR2c it must be shown that $C \cdot \theta$ trusts the instance $C\{y \mapsto u\} \cdot \theta$ of $(C[\![T]\!])\{y \mapsto u\}$. To this end, condition Trust2 is used. Let x be a variable occurring in some literal $L \in C\{y \mapsto u\}$ and let K be a literal in C such that $K\{y \mapsto u\} = L$. Using $\sigma = \{y \mapsto u\}$, it holds that $z\theta = z\sigma\theta$ for all variables z in C, because $C \cdot \theta$ is a ground instance of $C[\![y \equiv u, T]\!]$. Additionally, $K\sigma = L$ implies $L \preceq K\sigma$. □

Another prominent simplification rule is subsumption, which can be stated for the current context as follows.

$$\frac{C \qquad C\sigma \vee D[\![T]\!]}{C}\ \textsc{Subsumption}$$

1. if D is empty, then $C\sigma[\![T]\!] \sqsupset_{C\sigma\theta} C$ for all grounding substitutions θ, where $T\theta$ is true.

To show that Subsumption can be justified by simple clause redundancy, $C\sigma \vee D[\![T]\!]$ has to be simply redundant w.r.t. C, because only $C\sigma \vee D[\![T]\!]$ is deleted by Subsumption. Thus, let $C\sigma \vee D \cdot \theta$ be a ground instance of $C\sigma \vee D[\![T]\!]$. The proof proceeds by a case split on D.

- If D is nonempty, condition SCR1 is applied, where $C \cdot \sigma\theta$ is used for $D_1 \cdot \rho_1$. The clause $C\sigma\theta$ is a proper subclause of $(C\sigma \vee D)\theta$ and hence $\mathcal{F}(\mathcal{P}(C \cdot \sigma\theta)) \models \mathcal{F}(\mathcal{P}(C\sigma \vee D \cdot \theta))$, which shows condition SCR1a, and $C \cdot \sigma\theta \prec C\sigma \vee D \cdot \theta$, which shows condition SCR1b. It remains to prove condition SCR1c, i.e., $C\sigma \vee D \cdot \theta$ trusts the instance $C \cdot \sigma\theta$ of C. Here, Lemma 5 can be applied.

- If D is empty, condition SCR2 is applied, where $C \cdot \sigma\theta$ is used for $D \cdot \rho$. Condition SCR2a holds trivially and condition SCR2b holds because of condition 1 of Subsumption. Finally, condition SCR2c is fulfilled Lemma 5. □

The last simplification rule that is presented here is demodulation, which, given a unit equality $t \approx t'$ allows replacing an instance $t\sigma$ in a clause C by $t'\sigma$ and deleting the old clause, provided that $t\sigma \succ t'\sigma$. In our context, this rule is stated as follows.

$$\frac{t \approx t' \qquad C\langle t\sigma\rangle[\![T]\!]}{t \approx t' \qquad C\langle t'\sigma\rangle[\![T]\!]}\ \textsc{Demodulation}$$

1. $t\sigma \succ t'\sigma$;
2. $C\langle t\sigma\rangle \succ (t \approx t')\sigma$.

Since this proof is more involved than the ones before, a sketch is presented. We want to show that $C\langle t\sigma\rangle[\![T]\!]$ is simply redundant w.r.t. $t \approx t'$ and $C\langle t'\sigma\rangle[\![T]\!]$. Let $C\langle t\sigma\rangle \cdot \theta$ be a ground instance of $C\langle t\sigma\rangle[\![T]\!]$. We apply condition SCR1, with $t \approx t'$ for $D_1[\![U_1]\!]$ and $C\langle t'\sigma\rangle[\![T]\!]$ for $D_2[\![U_2]\!]$. Let $t \approx t' \cdot \sigma\theta$ and $C\langle t'\sigma\rangle \cdot \theta$ be the correctly ground instances used. If t' contains variables that are not in t, then σ can be extended. Because the demodulation is applied to a green context, it can be shown that $\mathcal{F}(\mathcal{P}(\{t \approx t' \cdot \sigma\theta, C\langle t'\sigma\rangle \cdot \theta\})) \models \mathcal{F}(\mathcal{P}(\{C\langle t\sigma\rangle \cdot \theta\}))$. Moreover, the conditions of Demodulation imply that $t \approx t' \cdot \sigma\theta \prec C\langle t\sigma\rangle \cdot \theta$ and $C\langle t'\sigma\rangle \cdot \theta \prec C\langle t\sigma\rangle \cdot \theta$. For condition SCR1c, $C\langle t\sigma\rangle \cdot \theta$ trusts the ground instance $t \approx t' \cdot \sigma\theta$ of $t \approx t'$ by Lemma 5. For $C\langle t'\sigma\rangle \cdot \theta$ condition Trust2 is employed using the identity substitution for σ.

5.7 Refutational Completeness

This section presents a sketch for refutational completeness of the constraint superposition calculus w.r.t. Henkin semantics. As in the previous completeness proofs, the result from the previous layer is used in each case, where the base case is the first-order calculus on the level PF. The results of Section 4.1 regarding the presented superposition for first-order logic with interpreted Booleans can not be used without modifications. The corresponding calculus for level PF is given in Figure 5.1. Term orders, eligibility and selection functions are defined as in Section 4.1, where instead of stating that a literal is eligible in a clause C w.r.t. to some substitution θ, the substitution is simply the one that is part of the closure $C \cdot \theta$. As for other calculi, it is required that closures for binary inference are variable-disjoint and thus the disjoint union of substitutions that are part of the closures in the premises is well-defined.

For refutational completeness of this calculus, the results of [Bac+92] and [NR92] need to be generalized to support the interpreted Boolean type. Several lemmata and definitions from [Num+21] can also be used in this case. Since the first-order version of λ-abstractions and De Bruijn indices are just function symbols, we need to distinguish terms that have a special syntactic property. To this end, we say that a first-order term t is *closed* if $\mathcal{F}^{-1}(t)$ is a valid term and not a preterm. If t' is a higher-order term without functional variables, then $\mathcal{F}(t')$ is always closed. Let $N_0 \subseteq N \subseteq \mathcal{C}_{\mathrm{PF}}$, where N_0 is the initial set of clauses and N is a saturated set of ground first-order closures obtained by exhaustively applying the inference rules above. If every term in N_0 is closed, it holds that also every term in N is closed because the inference rules always rewrite closed terms to closed terms. The fundamental result for level PF should then say that if N is saturated and $\bot \notin N$, then there is a rewrite system R_N that is a model of all the order-irreducible closures w.r.t. N, i.e., $R_N \models \mathrm{irred}_R(N)$. The term order for level PF can be obtained from level PG by defining $\mathcal{F}(t) \succ \mathcal{F}(t')$ if and only if $t \succ t'$ for $t, t' \in \mathcal{T}_{\mathrm{PG}}$.

Now, we consider level PG. To this end, let $N_0 \subseteq N \subseteq \mathcal{C}_{\mathrm{PG}}$, where N_0 is the initial clause set and N is a saturated set of ground closures and $\bot \notin N$. The used inference system is the one of the constraint superposition calculus, where for every inference rule the premises and the conclusions are mapped via $\mathcal{G}$ and $\mathcal{PS}$. Using the approach presented in Section 4.2.7, the first-order interpretation $R_{\mathcal{F}(N)}$ can be lifted to a higher-order interpretation $\mathcal{I}$ of all order-irreducible ground closures in N. Since, by assumption, every closure in N_0 is of the form $C \cdot \mathsf{id}$ for the identity substitution id, we have that every closure in N_0 is order-irreducible and therefore $\mathcal{I} \models N_0$. This property holds because the initial clause set on the nonground higher-order level is assumed to contain no constraints.

For the refutational completeness of level G, the inferences have to be lifted from level PG to level G. As before, let $N_0 \subseteq N \subseteq \mathcal{C}_{\mathrm{G}}$, where N_0 is the initial clause set and N is a saturated set of ground closures and $\bot \notin N$. Care has to be taken for the handling of functional variables. The calculus for level G is a translation of the constraint superposition calculus via the grounding function $\mathcal{G}$. If all inferences can be shown to be liftable or redundant, the result of the level PF gives a model $\mathcal{I}$ of $\mathcal{PS}(N_0)$ which can

$$\frac{(\overbrace{D' \vee t \approx t'}^{D}) \cdot \theta_1 \quad C[u] \cdot \theta_1}{(D' \vee C[t']) \cdot (\theta_1 \uplus \theta_2)}\ \textsc{Sup} \qquad \frac{(\overbrace{C' \vee u' \approx v' \vee u \approx v}^{C}) \cdot \theta}{(C' \vee v \not\approx v' \vee u \approx v') \cdot \theta}\ \textsc{Factor}$$

$$\frac{(\overbrace{C' \vee u \not\approx u'}^{C}) \cdot \theta}{C' \cdot \theta}\ \textsc{Irrefl} \qquad \frac{(\overbrace{C' \vee s \approx t}^{C}) \cdot \theta}{C' \cdot \theta}\ \bot\textsc{Elim}$$

$$\frac{C[u] \cdot \theta}{C[t'] \cdot \theta}\ \textsc{BoolRw} \qquad \frac{C[u] \cdot \theta}{(C[\bot] \vee u \approx \top) \cdot \theta}\ \textsc{BoolHoist}$$

$$\frac{C[s \mathbin{\approx\!\!\!\approx} t] \cdot \theta}{(C[\bot] \vee s \approx t) \cdot \theta}\ \approx\textsc{Hoist} \qquad \frac{C[s \not\approx t] \cdot \theta}{(C[\top] \vee s \approx t) \cdot \theta}\ \not\approx\textsc{Hoist}$$

The rules are subject to the side conditions given below:

Sup $t\theta_1 = u\theta_2$; u is not a variable; $t\theta' 1 \succ t'\theta_1$; $D\theta_1 \prec C[u]\theta_2$; the position of u is $\succeq$-eligible in $C \cdot \theta_2$; $t \approx t'$ is strictly $\succeq$-eligible in $D \cdot \theta_1$; the root of t is not a logical symbol; if $t'\theta_1 = \bot$, the subterm u is at the top level of a positive literal.

Factor $u\theta = u'\theta$; $u\theta \approx v\theta$ is $\succeq$-eligible in $C \cdot \theta$; $u\theta \succ v\theta$.

$\bot$Elim $(s \approx t)\theta = \bot \approx \top$; $s \approx t$ is strictly $\succeq$-eligible in $C \cdot \theta$.

BoolRw condition 1 from BoolRw of Section 4.1.3 where the variable x is replaced by a placeholder term s; $u\theta = t$; u is not a variable; the position of u is $\succeq$-eligible in $C \cdot \theta$.

BoolHoist u is a Boolean term whose root is an uninterpreted predicate; the position of u is $\succeq$-eligible in C; u is not at the top level of a positive literal.

$\star$Hoist (where $\star \in \{\mathbin{\approx\!\!\!\approx}, \not\approx\}$) the position of the indicated subterm is $\succeq$-eligible in $C \cdot \theta$.

Figure 5.1: Inference Rules for level PF

be transformed into $\mathcal{I}'$ such that $\mathcal{I} \models N_0$ by correctly handling the functional variables mapped by the mapping $\mathcal{PS}$.

Finally, for level H and $N_0 \subseteq N \subseteq \mathcal{C}_{\mathrm{H}}$ such that $\bot \notin N$ and N is saturated w.r.t. the constraint superposition calculus, a model for N_0 has to be constructed. This is done by using the model $\mathcal{I}$ of $\mathcal{G}(N_0)$ which can be obtained by the results for level G. Thus, if all the concrete proof steps are valid, the result of refutational completeness holds for the constraint superposition calculus.

5.8 Saturation Procedure

The saturation procedure for the constraint superposition calculus is only a slight variation of Algorithm 2 and is given by Algorithm 5 below.

Algorithm 5 (Higher-order constrained given clause procedure)

function ExtractClause(Q, *stream*)
  maybe_clause ← pop and compute first element of *stream*
  if *stream* is not empty **then**
    add *stream* to Q with an increased weight
  return *maybe_clause*

function HeuristicProbe(Q)
  $(collected\_clauses, i) \leftarrow (\emptyset, 0)$
  while $i < K_{\text{best}}$ and Q is not empty **do**
    $(maybe\_clause, j) \leftarrow (\emptyset, 0)$
    while $J < K_{\text{retry}}$,Q is not empty, and $maybe\_clause = \emptyset$ **do**
      stream ← pop the lowest weight stream in Q
      maybe_clause ← ExtractClause(Q, *stream*)
      $j \leftarrow j + 1$
    $collected\_clauses \leftarrow collected\_clauses \cup maybe\_clause$
    $i \leftarrow i + 1$
  return *collected_clauses*

function FairProbe(Q, *num_oldest*)
  collected_clauses ← $\emptyset$
  oldest_streams ← pop *num_oldest* oldest streams from Q
  for *stream* in *oldest_streams* **do**
    collected_clauses ← *collected_clauses* ∪ ExtractClause(Q, *stream*)
  return *collected_clauses*

function ForceProbe(Q)
  collected_clauses ← $\emptyset$
  while Q is not empty and *collected_clauses* $= \emptyset$ **do**
    collected_clauses ← FairProbe(Q, $|Q|$)
  if Q and *collected_clauses* are empty **then** *status* ← Satisfiable
  else *status* ← Unknown
  return (*status*, *collected_clauses*)

```
function GivenClause(P)
  A ← ∅
  Q ← empty priority queue
  status ← Unknown
  i ← 0
  while status = Unknown do
    if P is not empty then
      given ← pop chosen clause from P and simplify it using A
      if given = ⊥⟦T⟧ such that all constraints in T are flex-flex then
        status ← Unsatisfiable
      else
        A ← A ∪ {given}
        for stream in streams of inferences between given and clauses in A do
          if stream is not empty then P ← P ∪ ExtractClause(Q, stream)
        i ← i + 1
        if i mod K_fair = 0 then P ← P ∪ FairProbe(Q, ⌊i/K_fair⌋)
        else P ← P ∪ HeuristicProbe(Q)
    else // P is empty
      (status, forced_clauses) ← ForceProbe(Q)
      P ← P ∪ forcedClauses
  return status
```

The only textual change is highlighted with a yellow background in the function GivenClause. There, instead of checking if the given clause is empty, the constraints have to be also checked. To this end, if the given clause contains no literals and all constraints are flex-flex the status Unsatisfiable can be returned because then a contradiction is found. Moreover, in an implementation one can decide if inferences between constrained clauses can be carried out at any time or if the constrained clauses should already be in a state where all the constraints of the involved clauses are flex-flex.

5.9 Implementation in Zipperposition

I implemented the constraint superposition calculus in the Zipperposition automated theorem prover [Cru15].

Its name is a pun that mixes the words *zipper* (a functional data-structure invented by Huet) and *superposition* (as in the superposition calculus). Because the superposition calculus described in Section 4.2 was already implemented, I was able to build upon this work. The overall goal is to add unification constraints to clauses and to be able to provide a configuration switch to only use the constraints if the switch is set.

Zipperposition is developed in the programming language OCaml [Ler+22]. OCaml supports functional, object-oriented, and imperative programming styles. Its rich type system allows the definition of (generalized) algebraic data types, modules and functors,

which can be thought of functions that transform modules to modules. In Zipperposition, these features are used extensively where they help to catch several errors at compile term which would otherwise occur as runtime errors in languages without sophisticated type systems. Although OCaml is a high-level language, the compiler emits reasonably fast executables. Zipperposition won the higher-order problem category of the CADE ATP System Competition in the years 2020, 2021, and 2022 [Sut20; Sut21; Sut22]. The project is developed such that new features can be easily implemented as extensions. This approach is used to define several different calculi inside Zipperposition. In the beginning, the prover only supported first-order logic. But by defining several extensions and due to the modular architecture of the project, each calculus leading to the higher-order superposition calculus and the constraint superposition calculus for higher-order logic is implemented in Zipperposition. Every supported calculus can be selected by its corresponding command-line argument.

The source code of Zipperposition is publicly available at GitHub under `https://github.com/sneeuwballen/zipperposition`. I forked this repository in order to add my changes. Hence, the source code discussed in this section and the version of Zipperposition used for the evaluation can be obtained at `https://github.com/hetzenmat/zipperposition`. In the following, I document how constraints were added to clauses. Moreover, I describe the necessary changes needed in order to turn the unification procedures of Zipperposition into preunification procedures. Finally, the altered saturation procedure is discussed, as well as new simplification rules.

5.9.1 Constrained Clauses

In order to add support for constrained clauses, I added a set of functions which operate on constraints, located in the file `Constraints.ml`. Here, a single constraint is just a pair of terms and a collection of constraints, then, is a list of pairs of terms. This is modelled in OCaml as follows.

```
type elem = Term.t * Term.t

type t = elem list
```

Thus, other code can now use `Constraints.t` to refer to values that model constraints. Instead of a simple list, a more complex data structure would possibly be more performant for operations on constraints, but I chose this representation since this closely resembles our definition of constrained clauses. Now, the constraints can simply be added to the data type representing clauses. This is done via the following type defined in the file `Constraints.ml`, where I just had to add the field called `constraints`.

```
type t = {
  id : int; (** unique ID of the clause *)
  lits : Literal.t array; (** the literals *)
  trail : Trail.t; (** boolean trail *)
  constraints : Constraints.t; (** unification constraints *)
  mutable flags : flag; (** boolean flags for the clause *)
}
```

Hence, a clause is modelled by the above record type. The record field `trail` is used for Zipperposition's AVATAR implementation for clause splitting [Vor14]. For the constraint superposition calculus, the extension implementing clause splitting was disabled and thus for the implementation of the constraint superposition calculus, the record field `trail` can be disregarded.

When checking if a clause is unsatisfiable, the constraints have to be respected in the following function.

```
let is_empty c =
    Literals.is_absurd c.lits
    && Trail.is_empty c.trail
    && List.for_all Constraints.is_flex_flex c.constraints
```

This function checks if each literal is absurd, that is, if it is either the constant $\bot$ or an equation of the form $t \not\approx t$. If there are no literals in the clause, the expression `Literals.is_absurd c.lits` is also true. After the trail is checked, if all constraints are flex-flex it can be safely reported that the given clause is empty.

5.9.2 Preunification

The unification procedures implemented in Zipperposition operate like the ones presented in this thesis. That is, the current state of a unification problem is modelled as a list of pairs of terms, which represent the constraints that need to be solved, and a substitution, which is the partial solution constructed this far. At each unification step, it first is checked if the list of unification constraints that need to be solved is empty. If this is the case, the current partial solution is also a solution to the whole problem. Otherwise, the remaining cases need to be checked. The main unification loop is implemented in the file `UnifFramework.ml`.

To also support preunification, I introduce the following code before all other cases are checked.

```
match problem with
| [] -> (* no constraints, found a solution *)
    OSeq.return (Some (Unif_subst.of_subst subst))
| _ when P.preunification && all_flex_flex problem ->

    let p = ref problem in
    let sub = ref subst in

    while !p != [] do
        let ((lhs, rhs), tail) = Future.head_tail !p in
        p := tail;

        match unif_types !sub lhs rhs with
        | None -> raise Unif.Fail
        | Some subst ->
            sub := subst;
    done;

    let unifier =
        Unif_subst.make !sub (make_constraints problem)
    in
    OSeq.return (Some unifier)
| (lhs, rhs) as current_constraint :: rest ->
  (* remaining cases *)
```

This code is rather imperative, but could also be refactored into a more functional style using `List.fold`. The intention here is to return a solution early if all constraints are flex-flex. If this is the case, it remains to unify also all the types of the flex-flex pairs. If for some flex-flex pair the types are not unifiable w.r.t. to the current substitution, then the `Unif.Fail` exception is raised, which signals that there is no solution for the current branch. Otherwise, a substitution is returned where the remaining flex-flex pairs are added as constraints. This branch of the `match` construct is only taken if the flag `P.preunification` is set. Using the functional iterator module `OSeq`, the solutions of the unification problems can be computed by demand.

5.9.3 Saturation Procedure

Zipperposition's saturation procedure is a slight variation of Algorithm 5. The proof state is stored in the environment module implemented in `env.ml`. Using the function `next_passive : unit -> Clause.t option`, the next passive clause is chosen if there is one. The function implementing a single step of the given clause procedure, realized in `saturate.ml`, looks slightly simplified as follows.

```
let given_clause_step  num =
    (* select next given clause *)
    match Env.next_passive () with
    | None ->
        (* final check: might generate other clauses *)
        let clauses = Env.do_generate () in
        if Iter.is_empty clauses
        then Sat
        else (
            Env.add_passive clauses;
            Unknown
        )
    | Some c when not (Env.C.only_flex_flex c) ->
    (*
    Clause has non flex-flex constraints
    Send corresponding signal and do nothing else;
    this case should be handled in the Superposition module
    *)

    Signal.send
        Env.on_given_clause_with_non_flex_flex_constraints c;
    Unknown
    | Some c ->
    (* check if c is the empty clause *)
    (* otherwise, perform generating inferences *)
```

If a given clause is encountered that as non flex-flex constraints, the corresponding signal is sent and the status `Unknown` is returned. In all other cases, no changes are necessary. This signal is then handled in the module implementing superposition calculi. When this signal is encountered, the constraints of the clause are used as the current state of a preunification problem. This problem is then added to the environment as a generating rule by inserting a new stream that produces clauses. Solutions to this preunification problem are then added to the passive clause set by calls to the function `Env.do_generate`.

5.9.4 Simplification Rules

To efficiently solve unification constraints, it can be attempted to solve individual constraints using unification procedures for specific subclasses of higher-order unification that always admit a most-general unifier. To this end, the following function is defined in `Constraints.ml`.

```
let try_unif (l,r) =
    [ (try_lfho_unif, "lfho")
    ; (try_fixpoint_unif, "fixpoint")
    ; (try_pattern_unif, "pattern") ]
    |> List.find_map (fun (alg,n) -> match alg l r with
                                     | Ok s -> Some (s, n)
                                     | Error _ -> None)
```

This function attempts to solve a unification constraint by using either λ-free higher-order unification, fixpoint unification, or pattern unification. If one of these procedures yields a most-general unifier it is returned.

Using this function, a simplification rule that performs partial unification of constraints is implemented, which is an instance of the Unif simplification rule. This rule calls the function `try_unif` for all constraints of a clause. If a constraint yields a substitution, the constraint is removed from the clause and the substitution is applied to the whole constrained clause.

The rule discussed above handles the case of unifiable constraints. The other case is when a constraint is evidently unsolvable. This is handled by a rule, which allows deleting clauses from the search space if the corresponding constraints admit no solution. Its implementation looks as follows.

```
let unsolvable constraints =
    let check (l,r) = different_rigid_rigid (l,r) ||
          ([ try_lfho_unif
           ; try_fixpoint_unif
           ; try_pattern_unif ]
           |> List.exists (fun alg -> match alg l r with
                                      | Ok _ -> false
                                      | Error b -> b))
    in
    List.exists check constraints
```

The constraints of a clause are unsolvable, if there is a constraint that is rigid-rigid. This case is checked by a call to the function `different_rigid_rigid`. Then, it is checked if one of the aforementioned unification procedures indicates that the problem has no solutions. This is done via the result `Error b`, where the Boolean `b` is true if the problem admits no solution. If the Boolean is false, this result just represents the case that the problem does not lie in the supported fragment.

5.10 Evaluation

The evaluation was performed on a server with an AMD EPYC 7502 32-core processor clocked at 2.50 GHz, 1 TB RAM, and 960 GB SSD. As in [Ben+23b], all 2606 TH0 theorems from the TPTP 7.3.0 library [Sut17] and 1253 „Judgment Day“ problems [BN10]

	TPTP	ofSH	SH	Total
Zipperposition	975	269	254	1498
Zipperposition with constraints	973	269	255	1497

Figure 5.2: Evaluation against original Zipperposition

generated using Sledgehammer [PB10] were used as the benchmark set. The benchmarks and raw evaluation data is publicly available[1]. The wall-clock time limit for every problem was 30 seconds. In the following, the TH0 theorems from TPTP are denoted by „TPTP“. Each of the 1253 „Judgment Day“ problems comes in two variants. One variant uses the native Boolean type. These problems are denoted by „SH“. The other variant uses an encoding into Boolean-free higher-order logic, and the problems thereof are denoted by „ofSH“. Thus, the whole benchmark suite consists of 5112 problems. Zipperposition is a *cooperative theorem prover*. That is, a backend reasoner may be executed to try to finish the current proof attempt. For Zipperposition the backend reasoner is usually E. In all the experiments in this section, no backend reasoner was used in order to only measure the characteristics of Zipperposition itself.

In the first experiment, it was tested if the added implementation of unification constraints degrades performance when the unification constraints are disabled. Zipperposition implements several different modes that can be used for higher-order problems. The mode „complete“ enables all inference rules given in Section 4.2.5 and uses the complete and possibly non-terminating higher-order unification algorithm of [VBN20]. To this end, the latest official version of Zipperposition and my implementation were executed in the complete higher-order mode. Although my implementation is called „Zipperposition with constraints“, the handling of unification constraints was disabled. The results can be seen in Figure 5.2. Hence, my implementation does not impose a noticeable performance slowdown.

I implemented two modes for the constraint superposition calculus. The complete mode, called „constraints-complete“, enables all inferences rules given in Section 5.5 and uses the possibly non-terminating complete preunification algorithm described in Section 5.2. The optimized mode, called „constraints-best“, resembles the mode „best“ of Zipperposition. In the „best“ mode, explosive rules such as FluidSup and the Ext axiom are disabled, because they either introduce fresh higher-order variables or are able to transform first-order into higher-order problems. Moreover, the unification algorithm uses limits to cut off the search after a certain number of iterations to get a terminating but incomplete approximation of unifiers. Also, inference rules dealing with the native Boolean type are disabled. In the „constraints-best“ mode, also rules such as FluidSup and the Ext axiom were disabled. Moreover, Boolean reasoning is disabled and the preunification algorithm has an iteration limit.

[1] https://doi.org/10.5281/zenodo.8284039

	TPTP	ofSH	SH	Total
complete	973	269	255	1497
constraints-complete	1116	225	214	1555

Figure 5.3: Evaluation of „complete“ mode against „constraints-complete“ mode

	TPTP	ofSH	SH	Total
best	1822	455	447	2724
constraints-best	1453	319	301	2073

Figure 5.4: Evaluation of „best“ mode against „constraints-best“ mode

Thus, one experiment compares the modes „complete“ and „constraints-complete“ and another experiment compares the modes „best“ and „constraints-best“. The results are given by Figure 5.3 and Figure 5.3, respectively.

It can be seen that the „constraints-complete“ mode improves upon the „complete“ mode regarding TPTP problems but performs worse on the SH and ofSH problem sets. A possible reason for this behavior is that TPTP problems use more features of higher-order logic than problems generated using Sledgehammer from Isabelle, which is also noticed by Jasmin Blanchette in [Bla23]. The „constraints-best“ mode is a clear improvement compared to the „constraints-complete“ mode, as over 500 more problems could be solved. When comparing the results of the last experiment, it is evident that the „constraints-best“ mode is not competitive compared to the „best“ mode. A possible explanation is that the „best“ mode uses more fine-tuned heuristics than my „constraints-best“ mode. However, there are 103 problems that were solved by „constraints-best“ but not by „best“. In competitions, Zipperposition is executed in a portfolio of different configurations. Thus, the „constraints-best“ mode would be a possible candidate for addition to a portfolio of different configurations.

CHAPTER 6

Related Work

Besides the superposition calculus, there are many other approaches for higher-order theorem proving. Some of them rely on a translation into first-order logic, of which there are several different possible translations. One has to choose an approach for the elimination of λ-abstractions, for example combinators or λ-lifting, where new function definitions are added that replace the actual λ-expressions [Hug82]. Then, a regular first-order prover can be used and depending on the translation, the resulting proof has to be checked for soundness [Ker91].

The combinatory superposition calculus is used by Vampire [BR20]. Here, higher-order terms are encoded using combinators and the combinator axioms are used as rewrite rules. Moreover, since first-order unification is employed, the calculus should be easy to implement for existing first-order provers. By using a strategy schedule construction tool, Vampire succeeded in claiming the first place at the CADE ATP System Competition 2023 in the category of higher-order logic theorems [Sut23], overtaking Zipperposition.

A sequent calculus for higher-order logic is employed by the prover AgsyHOL [Lin14]. Using sequent calculi for automated provers is convenient because they are well suited for proof search. AgsyHOL tries to find a proof by applying the inference rules from the goal backwards one after another until a complete derivation tree is constructed.

The prover Satallax applies a tableau calculus which is navigated via a SAT solver [Bro12]. It uses a transformation from higher-order terms to propositional literals and clauses. Satallax starts with a conjunction of the given axioms together with the negation of the conjecture. Using a SAT solver and the mapping to propositional logic, the formula is tested for satisfiability. If the SAT solver reports unsatisfiability, also the original problem is unsatisfiable.

Also, the SMT provers CVC4 and veriT have been enhanced to support higher-order logic [Bar+19]. Two approaches are presented: The pragmatic one extends an SMT solver that is targeted for first-order cases only with a minimal number of modifications. While in

the redesign approach, the data structures and algorithms of the solver are redeveloped from the bottom up. For solvers with a smaller code base, the redesign approach is suggested and for complex solvers the pragmatic approach might work better.

CHAPTER 7

Conclusion

In this thesis, we have presented a novel superposition calculus for higher-order logic that employs unification constraints which was implemented in the state-of-the-art automated theorem prover Zipperposition.

In Chapter 3 a framework for saturation-based theorem proving introduced by Waldmann et al. [Wal+22] is discussed. By instantiating the framework for a given inference system and redundancy criterion, one is able to obtain refutational completeness of a calculus by lifting the result from the ground to the non-ground case. Moreover, the framework can be used for any suitable logic.

A variation of the classical superposition calculus is presented in Chapter 4. This calculus adds support for an interpreted Boolean type as well as inprocessing clausification. Its refutational completeness is used by the superposition calculus for higher-order logic. By using nonstandard models, i.e., Henkin semantics, refutational completeness can be obtained without being affected by Gödel's first incompleteness theorem.

The constraint superposition calculus was introduced in Chapter 5. By employing a simple version of clause redundancy, several simplification rules can be easily justified. An efficient preunification algorithm is presented, which allows cutting the search space when only variable-headed terms, i.e., flex-flex pairs, remain. Moreover, we give a proof sketch for refutational completeness. The implementation of the calculus in Zipperposition is documented, and its effectiveness empirically evaluated.

7.1 Future Work

There are several possibilities for future work. First and foremost, the constraint superposition calculus should be proven refutationally complete, which is already ongoing work. When having a pen-and-paper proof, a major result would be a formal proof using a proof assistant. An attractive proof assistant for this venture is Isabelle/HOL. The

reason for this is that Nicolas Peltier proved refutational completeness of a variant of the first-order superposition calculus in Isabelle/HOL [Pel16]. Also the saturation framework is fully formalized in Isabelle/HOL [Tou20; BT20]. Moreover, Blanchette et al. used Isabelle/HOL to formally verify variants of given clause procedures [BQT23], one of which is Zipperposition's given clause loop. From a formal proof, it then may be possible to extract code in a programming language such as OCaml, Haskell, or Scala in the case of Isabelle/HOL. Although extracted code from formal theorem provers tends to be rather slow, the resulting program would be correct by construction and could serve as a tool to catch soundness bugs in more performant implementations, since the extracted program is expected to give the correct answer.

Additionally, the implementation in Zipperposition can be improved by developing specialized heuristics and data structures. The new modes supporting unification constraints and preunification could be incorporated into the portfolio to check if this increases the performance of Zipperposition in competition contexts. Since the higher-order superposition calculus is implemented in the E theorem prover [VBS23], it would be possible to add support for the unification constraint approach, without first needing to implement the whole calculus.

Bibliography

[Bac+92] Leo Bachmair et al. „Basic Paramodulation and Superposition“. In: *CADE-11*. Ed. by Deepak Kapur. Vol. 607. LNCS. Springer, 1992, pp. 462–476.

[Bar+19] Haniel Barbosa et al. „Extending SMT Solvers to Higher-Order Logic“. In: *Automated Deduction – CADE 27*. Ed. by Pascal Fontaine. Cham: Springer International Publishing, 2019, pp. 35–54. ISBN: 978-3-030-29436-6.

[Ben+18] Alexander Bentkamp et al. „Superposition for Lambda-Free Higher-Order Logic“. In: *Automated Reasoning*. Ed. by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Cham: Springer International Publishing, 2018, pp. 28–46. ISBN: 978-3-319-94205-6.

[Ben+21] Alexander Bentkamp et al. „Superposition with Lambdas“. In: *Journal of Automated Reasoning* 65.7 (2021-08), pp. 893–940. DOI: 10.1007/s10817-021-09595-y.

[Ben+23a] Alexander Bentkamp et al. „Mechanical Mathematicians“. In: *Commun. ACM* 66.4 (2023-03), pp. 80–90. ISSN: 0001-0782. DOI: 10.1145/3557998.

[Ben+23b] Alexander Bentkamp et al. „Superposition for Higher-Order Logic“. In: *Journal of Automated Reasoning* 67.1 (2023-01). DOI: 10.1007/s10817-022-09649-9.

[Ben02] Christoph Benzmüller. „Comparing Approaches To Resolution Based Higher-Order Theorem Proving“. In: *Synthese* 133.1/2 (2002-10), pp. 203–235. DOI: 10.1023/a:1020840027781.

[Ben21] Alexander Bentkamp. „Superposition for Higher-Order Logic“. English. PhD thesis. Vrije Universiteit Amsterdam, 2021-05.

[Ben23] Alexander Bentkamp. *Errata of my PhD thesis „Superposition for Higher-Order Logic“*. 2023. URL: https://matryoshka-project.github.io/pubs/bentkamp_phd_thesis_errata.pdf (visited on 2023-05-05).

[BG01] Leo Bachmair and Harald Ganzinger. „Resolution Theorem Proving“. In: *Handbook of Automated Reasoning (in 2 volumes)*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 19–99. DOI: 10.1016/b978-044450813-3/50004-7.

[BG94] Leo Bachmair and Harald Ganzinger. „Rewrite-Based Equational Theorem Proving with Selection and Simplification“. In: *J. Log. Comput.* 4 (1994), pp. 217–247.

[BJR15] Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. „The Computability Path Ordering“. In: *Log. Meth. Comput. Sci.* 11.4 (2015).

[Bla23] Jasmin Blanchette. „λ-Superposition: From Theory to Trophy“. In: (2023). URL: https://matryoshka-project.github.io/pubs/trophy.pdf (visited on 2023-08-24).

[BN10] Sascha Böhme and Tobias Nipkow. „Sledgehammer: Judgement Day“. In: *Automated Reasoning*. Springer Berlin Heidelberg, 2010, pp. 107–121. DOI: 10.1007/978-3-642-14203-1_9.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. DOI: 10.1017/CBO9781139172752.

[BQT23] Jasmin Christian Blanchette, Qi Qiu, and Sophie Tourret. „Given Clause Loops“. In: *Archive of Formal Proofs* (2023-01). https://isa-afp.org/entries/Given_Clause_Loops.html, Formal proof development. ISSN: 2150-914x.

[BR20] Ahmed Bhayat and Giles Reger. „A Combinator-Based Superposition Calculus for Higher-Order Logic“. In: *Automated Reasoning*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Cham: Springer International Publishing, 2020, pp. 278–296. ISBN: 978-3-030-51074-9.

[Bro12] Chad E. Brown. „Satallax: An Automatic Higher-Order Prover“. In: *Automated Reasoning*. Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 111–117. ISBN: 978-3-642-31365-3.

[BT20] Jasmin Christian Blanchette and Sophie Tourret. „Extensions to the Comprehensive Framework for Saturation Theorem Proving“. In: *Archive of Formal Proofs* (2020-08). https://isa-afp.org/entries/Saturation_Framework_Extensions.html, Formal proof development. ISSN: 2150-914x.

[Cha11] Arthur Charguéraud. „The Locally Nameless Representation“. In: *Journal of Automated Reasoning* 49.3 (2011-05), pp. 363–408. DOI: 10.1007/s10817-011-9225-2.

[Coo03] S.B. Cooper. *Computability Theory*. Chapman Hall/CRC Mathematics Series. Taylor & Francis, 2003. ISBN: 9781584882374.

[CP03] Iliano Cervesato and Frank Pfenning. „A Linear Spine Calculus“. In: *J. Log. Comput.* 13 (2003), pp. 639–688.

[Cru15] Simon Cruanes. „Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond“. Theses. École polytechnique, 2015-09. URL: https://hal.science/tel-01223502.

[de 72] N.G de Bruijn. „Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem“. In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0.

[Fit02] Melvin Fitting. *Types, Tableaus, and Gödel's God*. Springer Netherlands, 2002. DOI: 10.1007/978-94-010-0411-4.

[Gol81] Warren D. Goldfarb. „The undecidability of the second-order unification problem“. In: *Theoretical Computer Science* 13.2 (1981), pp. 225–230. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(81)90040-2.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. USA: Cambridge University Press, 1989. ISBN: 0521371813.

[Hen50] Leon Henkin. „Completeness in the Theory of Types“. In: *The Journal of Symbolic Logic* 15.2 (1950), pp. 81–91.

[Hue72] Gérard Huet. „Constrained Resolution: A Complete Method for Higher-Order Logic.“ AAI7306307. PhD thesis. USA, 1972.

[Hue75] Gérard Huet. „A unification algorithm for typed λ-calculus“. In: *Theoretical Computer Science* 1.1 (1975), pp. 27–57. ISSN: 0304-3975.

[Hug82] R. J. M. Hughes. „Super-Combinators a New Implementation Method for Applicative Languages“. In: *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*. LFP '82. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1982, pp. 1–10. ISBN: 0897910826. DOI: 10.1145/800068.802129.

[JP76] D.C. Jensen and T. Pietrzykowski. „Mechanizing ω-order type theory through unification“. In: *Theoretical Computer Science* 3.2 (1976), pp. 123–171. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(76)90021-9.

[Ker91] Manfred Kerber. „How to Prove Higher Order Theorems in First Order Logic“. In: *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI'91. Sydney, New South Wales, Australia: Morgan Kaufmann Publishers Inc., 1991, pp. 137–142. ISBN: 1558601600.

[KMV11] Laura Kovács, Georg Moser, and Andrei Voronkov. „On Transfinite Knuth-Bendix Orders“. In: *Automated Deduction – CADE-23*. Ed. by Nikolaj Bjørner and Viorica Sofronie-Stokkermans. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 384–399. ISBN: 978-3-642-22438-6.

[Kőn27] Dénes Kőnig. „Über eine Schlussweise aus dem Endlichen ins Unendliche“. In: *Acta Sci. Math. (Szeged)* 3499/2009.3:2–3 (1927), pp. 121–130.

[KV13] Laura Kovács and Andrei Voronkov. „First-Order Theorem Proving and Vampire“. In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–35. ISBN: 978-3-642-39799-8.

[Ler+22] Xavier Leroy et al. *The OCaml system release 5.0. Documentation and user's manual*. 2022-12-20, pp. 1–989. URL: https://v2.ocaml.org/releases/5.0/ocaml-5.0-refman.pdf (visited on 2023-07-05).

[Lin14] Fredrik Lindblad. „A Focused Sequent Calculus for Higher-Order Logic“. In: *Automated Reasoning*. Ed. by Stéphane Demri, Deepak Kapur, and Christoph Weidenbach. Cham: Springer International Publishing, 2014, pp. 61–75. ISBN: 978-3-319-08587-6.

[LM21] Tomer Libal and Dale Miller. „Functions-as-constructors higher-order unification: extended pattern unification“. In: *Annals of Mathematics and Artificial Intelligence* 90.5 (2021-09), pp. 455–479. DOI: 10.1007/s10472-021-09774-y.

[LW07] Michel Ludwig and Uwe Waldmann. „An Extension of the Knuth-Bendix Ordering with LPO-Like Properties“. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Nachum Dershowitz and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 348–362. ISBN: 978-3-540-75560-9.

[Nip93] T. Nipkow. „Functional unification of higher-order patterns“. In: *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*. 1993, pp. 64–74. DOI: 10.1109/LICS.1993.287599.

[NR01] Robert Nieuwenhuis and Albert Rubio. „Chapter 7 - Paramodulation-Based Theorem Proving“. In: *Handbook of Automated Reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Amsterdam: North-Holland, 2001, pp. 371–443. ISBN: 978-0-444-50813-3. DOI: https://doi.org/10.1016/B978-044450813-3/50009-6.

[NR92] Robert Nieuwenhuis and Albert Rubio. „Basic Superposition is Complete“. In: *ESOP '92*. Ed. by Bernd Krieg-Brückner. Vol. 582. LNCS. Springer, 1992, pp. 371–389.

[Num+21] Visa Nummelin et al. *Superposition with First-Class Booleans and Inprocessing Clausification (Technical Report)*. Technical report. https://matryoshka-project.github.io/pubs/boolsup_report.pdf. 2021.

[Oka99] Chris Okasaki. *Purely Functional Data Structures*. en. Cambridge, England: Cambridge University Press, 1999-06.

[PB10] Lawrence C. Paulson and Jasmin Christian Blanchette. „Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers“. In: *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*. Ed. by Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska. Vol. 2. EPiC Series in Computing. EasyChair, 2010, pp. 1–11. DOI: 10.29007/36dt.

[Pel16] Nicolas Peltier. „A Variant of the Superposition Calculus“. In: *Archive of Formal Proofs* (2016-09). https://isa-afp.org/entries/SuperCalc.html, Formal proof development. ISSN: 2150-914x.

[Pre98] Christian Prehofer. *Solving higher order equations: from logic to programming*. Birkhäuser, 1998. ISBN: 978-3-7643-4032-2. URL: https://d-nb.info/952799111.

[Rob65] J. A. Robinson. „A Machine-Oriented Logic Based on the Resolution Principle“. In: *J. ACM* 12.1 (1965-01), pp. 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253.

[SCV19] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. „Faster, Higher, Stronger: E 2.3“. In: *Proc. of the 27th CADE, Natal, Brasil*. Ed. by Pascal Fontaine. LNAI 11716. Springer, 2019, pp. 495–507.

[SG89] Wayne Snyder and Jean Gallier. „Higher-order unification revisited: Complete sets of transformations“. In: *Journal of Symbolic Computation* 8.1 (1989), pp. 101–140. ISSN: 0747-7171. DOI: https://doi.org/10.1016/S0747-7171(89)80023-9.

[Smu11] Raymond Smullyan. *What is the Name of this Book?: The Riddle of Dracula and Other Logical Puzzles*. Dover Math Games & Puzzles. Dover Publications, Incorporated, 2011. ISBN: 9780486481982.

[Sut17] Geoff Sutcliffe. „The TPTP Problem Library and Associated Infrastructure“. In: *Journal of Automated Reasoning* 59.4 (2017-02), pp. 483–502. DOI: 10.1007/s10817-017-9407-7.

[Sut20] Geoff Sutcliffe. „Proceedings of the 10th IJCAR ATP System Competition (CASC-J10)“. In: (2020). URL: https://www.tptp.org/CASC/J10/Proceedings.pdf (visited on 2023-07-05).

[Sut21] Geoff Sutcliffe. „Proceedings of CASC-28 - the CADE-28 ATP System Competition“. In: (2021). URL: https://www.tptp.org/CASC/28/Proceedings.pdf (visited on 2023-07-05).

[Sut22] Geoff Sutcliffe. „Proceedings of the 11th IJCAR ATP System Competition (CASC-J11)“. In: (2022). URL: https://www.tptp.org/CASC/J11/Proceedings.pdf (visited on 2023-07-05).

[Sut23] Geoff Sutcliffe. „Proceedings of CASC-29 - the CADE-29 ATP System Competition“. In: (2023). URL: https://www.tptp.org/CASC/29/Proceedings.pdf (visited on 2023-07-19).

[Tou20] Sophie Tourret. „A Comprehensive Framework for Saturation Theorem Proving“. In: *Archive of Formal Proofs* (2020-04). https://isa-afp.org/entries/Saturation_Framework.html, Formal proof development. ISSN: 2150-914x.

[VBN20] Petar Vukmirović, Alexander Bentkamp, and Visa Nummelin. „Efficient Full Higher-Order Unification“. In: *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Ed. by Zena M. Ariola. Vol. 167. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 5:1–5:17. ISBN: 978-3-95977-155-9. DOI: 10.4230/LIPIcs.FSCD.2020.5.

[VBS23] Petar Vukmirović, Jasmin Blanchette, and Stephan Schulz. „Extending a High-Performance Prover to Higher-Order Logic“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Cham: Springer Nature Switzerland, 2023, pp. 111–129. ISBN: 978-3-031-30820-8.

[Vor14] Andrei Voronkov. „AVATAR: The Architecture for First-Order Theorem Provers“. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 696–710. ISBN: 978-3-319-08867-9.

[Vuk+21] Petar Vukmirović et al. „Making Higher-Order Superposition Work“. In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 415–432. ISBN: 978-3-030-79876-5.

[Wal+22] Uwe Waldmann et al. „A Comprehensive Framework for Saturation Theorem Proving“. In: *J. Autom. Reason.* 66.4 (2022), pp. 499–539. DOI: 10.1007/s10817-022-09621-7.