Implementation of Lambda-Free Higher-Order Superposition

MASTER'S THESIS

by

Petar Vukmirović

submitted to obtain the degree of

MASTER OF SCIENCE (M.SC.)

at

VRIJE UNIVERSITEIT AMSTERDAM
FACULTY OF SCIENCE

Course of Studies
COMPUTER SCIENCE

First supervisor:	Dr. Jasmin Christian BLANCHETTE Vrije Universiteit Amsterdam
Second supervisor:	Prof. Dr. Stephan SCHULZ DHBW Stuttgart

Amsterdam, February 2018

Abstract

In the last decades, first-order logic (FOL) has become a standard language for describing a large number of mathematical theories. Numerous proof systems for FOL which determine what formulas are universally true emerged over time. On the other hand, higher-order logic (HOL) enables one to describe more theories and to describe existing theories more succinctly. Due to more complicated higher-order proof systems, higher-order automatic theorem provers (ATPs) are much less mature than their first-order counterparts. Furthermore, many HOL ATPs are not effectively applicable to FOL problems. In this thesis, we extend E, a state-of-the-art first-order ATP, to a fragment of HOL that is devoid of lambda abstractions (LFHOL). We devise generalizations of E's indexing data structures to LFHOL, as well as algorithms like matching and unification. Furthermore, we generalized internal structures used by E as well as inferences and simplifications to support HOL features in an efficient manner. Our generalizations exhibit exactly the same behavior and time complexity as original E on FOL problems.

Acknowledgements

I would like to thank my supervisor, Jasmin Blanchette for patiently guiding me thorugh this project for the last couple of months. His support and know-how helped me write much better text and do better research than I could on my own. Furthermore, I would like to thank Stephan Schulz for providing me with valuable information on E implementation details, reading this thesis and giving very useful comments.

Many readers helped shape the text of the thesis further. In particular, I thank Ahmed Bhayat, Alexander Bentkamp, Alexander Steen, Daniel El Ouraoui, Giles Reger, Hans-Jörg Schurr, Pascal Fontaine, Predrag Janičić, Simon Cruanes and Tomer Libal for the time they took to read the thesis and make some very constructive comments.

I am specially thankful to the whole group of Theoretical Computer Science at Vrije Universiteit Amsterdam for making a pleasent atmosphere to work in. I would like to thank my family for support and especially friends Jelisaveta, Tara and Verica for listening through my stress rants on a daily basis. Last but not least, I thank Martijn for the understanding he has and taking every step in this journey with me for the last couple of months.

Contents

Chapter 1

Introduction

This thesis solves the problem of extending a first-order theorem prover to a fragment of higher-order logic. When choosing the fragment of higher-order logic there are many decisions to be made that can influence feasibility of the extension. In this chapter we motivate our choice and explain why it is useful and reasonable.

1.1 Motivation

In the last decades, first-order logic (FOL) has become a standard language for describing a large number of mathematical theories. Aside from its expressive power, numerous proof systems have been developed for first-order logic which facilitate determining sentences (formulas) that are universally true (valid).

Furthermore, a lot of effort has been put into automatically testing the validity of first-order logic sentences. This is important since it puts the stress on describing and formalizing mathematical theories and leverages the power of computer to do the hard work of proving. A tool that performs this kind of reasoning is called an automatic theorem prover.

Automatic theorem provers (ATPs) are used in both academic and commercial environments. In the former, ATPs have been able to solve long-open mathematical problems [McC97], whereas in the latter they are now routinely used in both software and hardware verification.

However, ATPs have their limits. They can be slow for many hard problems, and it is a fundamental result of theoretical computer science that there is no algorithm that decides formula's validity in all cases. For the second problem there is no real remedy, but the first problem has been tackled by using various proof calculi, data structures and programming languages to speed up ATPs.

Unlike FOL, higher-order logic (HOL) allows one to quantify over functions and predicates. This gives more expressive power to the mathematician formalizing a theory. Unfortunately, proof systems for HOL are much more complicated than the ones for FOL and their mechanization is immature compared to first-order logic.

Furthermore, there is no higher-order ATP that behaves as good as state-of-the-art first-order ATPs on the first-order problems [SBP13]. In other words, there might be translations of higher-order problems to FOL that are too hard for FOL provers or the original problems are too big for native higher-order provers.

For these reasons, it can be useful to create a theorem prover in which introduction of new features does not hinder its applicability on the problems that it could previously solve successfully. In particular, we want to extend first-order reasoners with higher-order capabilities, preserving their behavior on FOL problems.

To tackle proving theorems of HOL we decided to extend a first-order ATP to support higher-order reasoning instead of building a prover ground-up. In other words, the goal of this thesis is to answer a question:

Is there a way to extend a state-of-the-art first-order theorem prover to **HOL** *in a way that its* **performance on first-order problems remains the same?**

Such an extension will be called *graceful*. Before we tackle full HOL, we focused on a particular fragment of HOL.

The ATP that we chose is E [Sch02], state-of-the-art top contender in ATP competitions, that is also open source which enables us to customize it. It is fast for a large number of problems and carries the nickname of a *brainiac* theorem prover for its use of advanced data structures and sophisticated formula simplification schemes.

One of the requirements we had is that E has to perform almost exactly the same on FOL problems, which means that all of the data structures that E employs to speed up reasoning must have the same time complexity and all of the simplification techniques must be in place. Satisfying this requirement is not a simple task, since introduction of any feature in a highly-optimized prover may hinder its performance. However, our goal is to approach as little as 2% overhead on FOL problems. On the other hand, when faced with HOL features like partial application and applied variables it is unclear if one can gracefully generalize all E reasoning and simplification engines.

As a first step towards a graceful HOL reasoner we focused on fragment of HOL devoid of lambda-abstraction. The reason for this decision is that many problems can be stated without lambdas, and there are numerous translation techniques that can substitute lambdas with lambda-free formulas. However, this comes at the cost of losing deductive power, since we are unable to generate lambda terms during the proof search. Furthermore, lambda-free HOL terms are similar to FOL terms in many respects which allows us to reuse large parts of the E codebase unchanged. For the parts that must be generalized, we devised algorithms that gracefully generalize FOL counterparts. In the future, we might include lambdas in either native or translated form.

1.2 Contributions

To support applied variables and partial application we have generalized E's term structure and changed E's type system to support HOL types (Chapter 3). This generalization is entirely graceful, since the term structure is exactly the same for FOL terms and HOL features are supported in an efficient manner. Furthermore, following the recent work of Blanchette et al. on lambda-free HOL term orders [Bec+17]; [BWW17] we implemented a lambda-free HOL KBO order that is linear on both FOL and HOL terms.

Matching and unification are fundamental procedures used in automated theorem proving. In HOL with lambdas, unification is not even decidable and is usually tackled inside the proof calculus. In Chapter 4 we give a generalization of matching and unification for lambda-free HOL that keeps the same complexity for FOL terms and is efficient for HOL terms. Namely, even though HOL terms have twice as many subterms as FOL terms, we devised matching and unification procedures that consider the same number of subterms as the FOL algorithms by attempting to match or unify not only entire terms but also prefixes.

E uses advanced indexing techniques to speed up reasoning. These techniques were designed to be used with FOL terms and must be altered to support HOL features. In Section 5.1 we show a graceful extension of perfect discrimination trees, which is one of key data structures used in E. Like perfect discrimination trees, fingerprint indexing is used in various parts of the proof search. We extend it gracefully to HOL in Section 5.2.1. Lastly, graceful extension of feature vector indexing is described in Section 5.3.

One of the reasons E performs so well on first-order problems is the choice of the superposition calculus and the choice of simplification techniques. In Chapter 6 we gracefully generalize all the core calculus and simplification inferences. Some of the inferences involve traversing subterms of a term appearing in a formula. We managed to avoid traversing prefix subterms that appear only in HOL, keeping the time complexity for HOL inferences the same as in the FOL case.

1.3 Software

The main artefact of this project, next to this thesis, is a theorem prover for lambda-free HOL – *hoE*. This prover is based on E 2.1pre005 from October 2017.

hoE is highly experimental and is currently in the proof-of-concept phase. The code is undergoing constant changes, improvements and bugfixes. A thorough code review is planned for near future and it is expected that many procedures will change names or interfaces.

The repository in which E is kept is available for read-only access at `https://petarvukmirovic@bitbucket.org/petarvukmirovic/hoe.git`.

Chapter 2

Background

Even though first-order logic has become a standard when it comes to formalizing mathematical theories, there is still some non-standard notation that circulates in the literature. Furthermore, there are many interpretations of higher-order logic and it is very important to state clearly which stand we take in this thesis. To that end, in this chapter we define the notation that we are going to use thoughout the thesis and describe the assumed interpretations.

2.1 First-Order Logic

First order logic (FOL) is a language that is commonly used to model objects and their relations. It is an expressive language with well developed theory that makes it possible to reason about wide variety of objects – from everyday reasoning about family relations, over puzzle solving up to reasoning about complex mathematical structures and software verification.

Since it provides more expressiveness and it is commonly used in modern theorem provers, we will be concerned with many-sorted first-order logic with equality. In what follows, we will formally define the syntax and the semantics of such a language.

FOL is used as a meta-language in which we can describe a large number of theories. It consists of a logic part that includes connectives and quantifiers and a non-logic part that enables us to model different kinds of theories. We will assume that non-logic symbols are defined over the *signature* S, $S = (F, P, T)$. Signature defines the function symbols (F) and predicate symbols (P) and the structure $T = (A, rank_F, rank_P)$ that gives information about the types and arities of function symbols. A contains elementary types, whereas $rank_F : F \rightarrow A^* \times A$ declares function's type (and implicitly, arity) and $rank_P : P \rightarrow A^*$ declares predicate's type (and implicitly, arity). Lastly, for each atomic type a there is a countably infinite set of variables V_a.

Definition 1. Terms are defined inductively as follows:

1. A variable $x_a \in V_a$ is a term of type a.
2. If $t_1, \ldots, t_n$ are terms of types $a_1, \ldots, a_n$ respectively and $f \in F, rank_F(f) = (a_1 . \ldots . a_n, b)$, then $f(t_1, \ldots, t_n)$ is a term of type b.

If function symbol f has arity 0, we call it a *constant*, and drop the parentheses. Related to the notion of term is the notion of *subterm*. The set of subterms for term $t \equiv f(t_1, t_2, \ldots, t_n)$ consists of t itself, and all subterms of $t_1, t_2, \ldots, t_n$. A term is called *ground* if it is variable-free.

Definition 2. Term *position* is a (posibly empty) string of integers. Given a position p and a term t, we define term $t|_p$, that is subterm of t at position p recursively as follows:

$$t|_p = \begin{cases} t & \text{if } p \text{ is the empty string} \\ t_i|_{p'} & \text{if } t \equiv f(t_1, \ldots, t_n), p = i.p', i \geq 1, i \leq n \end{cases}$$

Similarly, we define changing subterm of t at position p for a term s ($t[p \leftarrow s]$) as follows:

$$t[p \leftarrow s] = \begin{cases} s & \text{if } p \text{ is the empty string} \\ f(t_1, \ldots, t_i[p' \leftarrow s], \ldots, t_n) & \text{if } t \equiv f(t_1, \ldots, t_n), p = i.p', i \geq 1, i \leq n \end{cases}$$

Definition 3. Atomic formulas (or atoms) are defined using the following two rules:

1. If $p \in P$, $rank_P(p) = a_1 . \ldots . a_n$ and $t_1, \ldots, t_n$ are terms of types $a_1, \ldots, a_n$ respectively then $p(t_1, \ldots, t_n)$ is an atom.

2. If s and t are two terms of the same type a, then $s \approx t$ is an atom.

Definition 4. Formulas are defined inductively as follows:

1. $\top$ and $\bot$ are formulas.

2. An atom is a formula.

3. If A is a formula, then $\neg A$ is a formula.

4. If A and B are formulas, then $A \land B$, $A \lor B$, $A \longrightarrow B$, $A \longleftrightarrow B$ are formulas.

5. If $x_a \in V_a$ and A is formula then $\forall x_a. A$ is a formula.

6. If $x_a \in V_a$ and A is formula then $\exists x_a. A$ is a formula.

Example 1. Now that we have formally defined the syntax of the first order logic with equality, we can introduce an involved example that highlights most of the features of the defined language. The signature we will use is defined as $S = (\{j, p, a\}, \{sibling, parent\}, T)$, where $T = (\{i\}, rank_F, rank_P)$ and $rank_F(j) = rank_F(p) = rank_F(a) = (\varepsilon, i)$, $rank_P(parent) = rank_P(sibling) = (i, i)$.

Suppose we want to model simple family relation: P and J are A's children. Does J have a sibling? One possible modeling is as follows:

$$\begin{gathered} ((parent(a, p) \land parent(a, j) \land \neg(p \approx j)) \\ \land (\forall x_i. \forall y_i. \forall z_i. (parent(x_i, y_i) \land parent(x_i, z_i) \land \neg(y_i \approx z_i)) \longrightarrow sibling(y_i, z_i)) \\ \land (\forall x_i. \forall y_i. sibling(x_i, y_i) \longrightarrow sibling(y_i, x_i))) \\ \longrightarrow \exists x_i. sibling(j, x_i). \end{gathered}$$

In the following sections we will describe how we can prove formulas such as the one from the above example, and even give answers to existential questions.

So far, we have been concerned with only well-formed formulas of FOL. We haven't discussed their meaning. There are several approaches to interpreting formulas of FOL, but we focus on standard one (given for example in book by Chang and Lee [CL73]), adapting it for many-sorted FOL.

An interpretation $I = (D, I_D, \nu)$ for signature $S = (F, P, T)$, $T = (A, rank_F, rank_P)$ is defined as follows. Let D be a family of non-empty disjoint sets indexed by $a \in A$ (called domains). I_D is a mapping from function and predicate symbols to functions and predicates and it is defined as follows: If $f \in F$, with $rank_F(f) = (a_1 \cdot \ldots \cdot a_n, b)$, then $I_D(f) = \mathbf{f}$, where $\mathbf{f} : D_{a_1} \times \ldots \times D_{a_n} \to D_b$. If $p \in P$, with $rank_P(f) = (a_1 \cdot \ldots \cdot a_n)$, then $I_D(p) = \mathbf{p}$, where $\mathbf{p} : D_{a_1} \times \ldots \times D_{a_n} \to \{\mathbf{T}, \mathbf{F}\}$, where $\mathbf{T}$ and $\mathbf{F}$ are designated constants interpreted as true and false. ν is a family of variable valuations indexed by $a \in A$. Each ν_a maps set of variables V_a to domain D_a. With $\nu_a[x_a := c_a]$ we label the valuation that is exactly the same as ν_a with the only (possible) difference that mapping of variable x_a is forced to be $c_a \in D_a$. In the following definitions we will assume an interpretation I, for signature S.

Definition 5. Interpretation of FOL terms is defined as mapping I_T from terms to domain corresponding to the type of the term, based on an interpretation I. I_T is defined recursively using the following rules:

1. If x_a is a variable of atomic type a, then $I_T(x_a) = \nu_a(x_a)$.
2. If c is a constant, then $I_T(c) = I_D(c)$
3. If $t_1, \ldots, t_n$ are terms, with interpretations $(\mathbf{t_1}, \ldots, \mathbf{t_n}) = (I_T(t_1), \ldots, I_T(t_n))$, then $I(f(t_1, \ldots, t_n))$ is $I_D(f)(\mathbf{t_1}, \ldots, \mathbf{t_n})$.

Definition 6. Interpretation of formulas of FOL is defined as mapping I_F from formulas to $\{\mathbf{T}, \mathbf{F}\}$, based on an interpretation I. This mapping is defined recursively using the following rules:

1. If $t_1, \ldots, t_n$ are terms with interpretation $(\mathbf{t_1}, \ldots, \mathbf{t_n}) = (I_T(t_1), \ldots, I_T(t_n))$ and $p \in P$, then $I_F(p(t_1, \ldots, t_n))$ is $I_D(p)(\mathbf{t_1}, \ldots, \mathbf{t_n})$.
2. If s and t are terms, then $I_F(s \approx t) = \mathbf{T}$ if and only if $I_T(s) = I_T(t)$ Otherwise $I_F(s \approx t) = \mathbf{F}$.
3. The logical connectives $\neg, \wedge, \vee, \longrightarrow, \longleftrightarrow$, are interpreted the same way as in propositional logic (for truth tables, consult [CL73]).
4. If x_a is of atomic type a, $I_F(\forall x_a.A) = \mathbf{T}$ if and only if for each $d_a \in D_a$, and interpretation $I' = (D, I_D, \nu[x_a := d_a])$, $I'_F(A) = \mathbf{T}$.
5. If x_a is of atomic type a, $I_F(\exists x_a.A) = \mathbf{T}$ if and only if for some $d_a \in D_a$, and interpretation $I' = (D, I_D, \nu[x_a := d_a])$, $I'_F(A) = \mathbf{T}$.

One can ask many interesting questions about a formula. For example, one could wonder if there is an interpretation in which a given formula is true. Furthermore, an interesting question is whether a formula is true in every interpretation. To be able to formally answer those questions we introduce the following definitions (following [CL73] closely):

Definition 7. Formula A is called *satisfiable* if there is an interpretation I such that $I_F(A) = \mathbf{T}$. In that case we call I a *model* for A and we say I *satisfies* A.

Definition 8. Formula A is called *unsatisfiable* if there is no interpretation I such that $I_F(A) = \mathbf{T}$.

Definition 9. Formula A is *valid* if for each interpretation I, $I_F(A) = \mathbf{T}$.

Definition 10. Formula A is a *logical consequence* of formulas $\{B_1, \dots, B_n\}$ if every interpretation I that is a model of all formulas $\{B_1, \dots, B_n\}$ is a model of A. If A is a logical consequence of $\{B_1, \dots, B_n\}$ we write $\{B_1, \dots, B_n\} \vDash A$

Definition 11. Formula A is *logically equivalent* to formula B if and only if under every interpretation, the truth values for A and B are the same.

In automated theorem proving, we will be usually interested with the question as to whether a given formula A (called *conjecture*) is a logical consequence of set of formulas S (called *axioms*). However, the method for proving that a formula is valid that we will be concerned with takes formulas in clause normal form (CNF). In what follows, we will define CNF and state basic results about it. The algorithm that transforms a formula to the CNF is out of the scope of this thesis, but is a very interesting topic on its own [RV01].

Definition 12. A formula A is in *prenex normal form* if and only if it is of the shape $Q_1 x1_{a_1} . Q_2 x2_{a_2} . \dots . Q_n x n_{a_n} . F$ where $Q_1, Q_2, \dots, Q_n$ are quantifiers and F is a quantifier-free formula. The sequence of quantifiers Q_i is called the prefix and formula F is called the matrix of formula A.

Definition 13. A literal l is an atomic formula or a negation of an atomic formula.

Definition 14. A formula A without quantifiers is in *conjunctive normal form* if and only if it is of the it is of shape $B_1 \wedge B_2 \wedge \dots \wedge B_n$ where each B_i is a disjunction of literals (or a *clause*).

Theorem 1. For each formula A there is a formula A' such that A is equivalent to A' and A' is in prenex normal form where matrix of A' is in conjunctive normal form.

Definition 15. Formula A is in *clause normal form* (CNF) if and only if it is of the shape $\forall x1_{a_1} . \forall x2_{a_2} . \dots . \forall x n_{a_n} . F$ where F is without quantifiers and in conjunctive normal form.

Alternatively, many proof procedures accept formulas in CNF as multiset of clauses, where each variable in a clause is implicitly universally quantified. It is clear that those two representation are equivalent since both $\wedge$ and $\vee$ are commutative and associative and the only quantifiers present are universal quantifiers.

We will not explain computing formula A' from Theorem 1 completely, but we will just hint that it is obtained by substituting subformulas of A for equivalent formulas that either take the quantifiers as much as possible to the left or that convert the matrix to conjunctive normal form.

Superposition works with sets of universally quantified clauses, that are interpreted as conjunction of their elements. With Theorem 1 we have approached this form very closely, but in the prefix of the formula we still might have existential quantifiers. A process called *Skolemization* creates a formula that is in the form needed by many automated theorem proving techniques.

A simple approach to Skolemization is removing the quantifiers from the left. Let the first existential quantifier in the prefix of formula A that is already in the prenex form with matrix in conjunctive normal form be $Q_i x i_{a_i}$ and $Q_1 x1_{a_1} . \dots . Q_{i-1} x i{-}1_{a_{i-1}}$ universal quantifiers that appear before it. We substitute $x i_{a_i}$ with a new term $sk_i(x1_{a_1}, \dots, x i{-}1_{a_{i-1}})$ where sk_i is a new function symbol of rank $(a_1 . \dots . a_{i-1}, a_i)$ and remove the quantifier $Q_i x_i$. We repeat this process until there is no existential quantifier left.

If we apply Skolemization to prenex formula A we indeed get the formula A' that is in CNF. But since we changed the signature and no longer apply equivalence-preserving transformations it is unclear what is the relation between original formula A and A' in terms of their models. Theorem 2 gives an answer to this question.

Theorem 2. For each formula A there is a formula A' in CNF that is satisfiable if and only if A is satisfiable.

In conclusion, after applying Skolemization we no longer have the formula that is equivalent to original one, but we do have the one that preserves satisfiability (and thanks to the fact that Theorem 2 is valid in both directions – unsatisfiability) of formula. This will be enough for proving the validity of a formula.

2.2 Refutational Theorem Proving

We will mostly be interested in the question as to whether a formula A logically follows from the set of formulas $\{B_1, B_2, \ldots, B_n\}$. However, thanks to duality in first-order logic a formula is valid if and only if its negation is unsatisfiable. This gives us a new way to approach the original question.

Moreover, we covered syntactic and semantic aspects of FOL, but we haven't touched upon deductive aspects, that is proof systems. To define proof systems, we need the notion of inference rules.

Definition 16. An inference rule is an $(n+1)$-ary relation between formulas $A_1, \ldots, A_n$ and A. Formulas A_1 up to A_n are called *premises* of the inference rule and formula A is called the *conclusion* of the rule. An inference rule is usually stated as:

$$\frac{A_1 \quad \ldots \quad A_n}{A}$$

The set of all conclusions where all of the premises belong to the set β is denoted as $\mathbf{C}(\beta)$. A *proof system* is a set of inference rules.

The question as to whether a formula A (called conjecture) is a logical consequence of set of formulas β (called axioms) corresponds to the question whether it can be proven from a set of formulas β.

Definition 17. A *proof* of a formula A from a set of formulas β is a sequence $P_1, \ldots, P_n$ where each P_i is either a formula from β or a conclusion of an inference rule whose premises belong to the set $\{P_j \mid j < i\}$. If there is a proof of A from β we write $\beta \vdash A$.

Definition 18. If there is a proof of a formula A from an empty set of axioms, then A is called *theorem*.

Validity of a formula is purely semantic notion – it is defined in terms of all possible interpretations of the formula. It is intuitively clear that exploring all possible interpretations is not feasible. Thus, one could wonder if it is possible to create a procedure that operates on a formula on a syntactic (deductive) level and still answers the interesting question as to whether a formula is true in all possible interpretations.

In refutational theorem proving, instead of proving that formula A is a valid, we show that A's negation is unsatisfiable. Similarly, instead of showing that A logically follows from $\{B_1, \ldots, B_n\}$ we show that set of formulas $\{B_1, \ldots, B_n, \neg A\}$ is unsatisfiable.

Definition 19. A proof system is *sound* if for every inference rule $\frac{A_1 \quad \dots \quad A_n}{A}$, $A_1 \land \dots \land A_n \vDash A$. Similarly, a proof system is *satisfiability preserving* if for all sets of formulas β, $\beta \cup \mathbf{C}(\beta)$ is satisfiable if β is satisfiable.

Definition 20. A proof system is *refutationally complete* if $\bot$ can be proven from any unsatisfiable set of formulas β.

Definition 21. A set of formulas β is saturated with respect to a proof system if and only if all possible conclusions of all possible inferences with premises from β are in β.

Intuitively, one would expect that for proof systems the minimal requirement is that they are sound. Relaxing this requirement to satisfiability preserving allows usage of useful techniques such as Skolemization. For refutational theorem proving it is enough that the system is satisfiability preserving. If the system is satisfiability preserving $\bot$ can be proven if and only if original set of formulas was inconsistent, which was our original goal.

The proof system that we will consider in this thesis (and many others) works with formulas in CNF and actually considers a formula as a (multi)set of clauses. Thus, it is important to have a way to transform original formula to multiset of clauses. Figure 2.1 shows how refutational theorem provers show that a formula is a theorem:

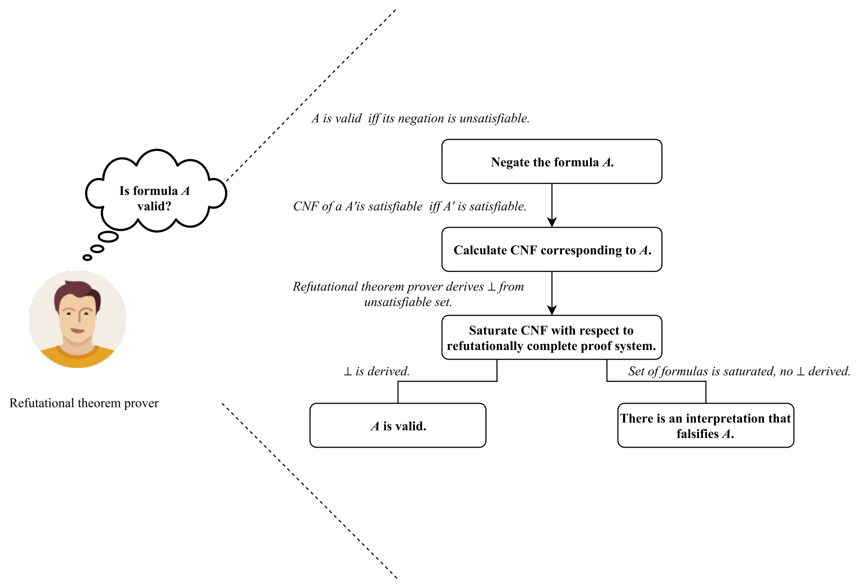
Is formula A valid?
Refutational theorem prover
A is valid iff its negation is unsatisfiable.
Negate the formula A.
CNF of a A'is satisfiable iff A' is satisfiable.
Calculate CNF corresponding to A.
Refutational theorem prover derives ⊥ from unsatisfiable set.
Saturate CNF with respect to refutationally complete proof system.
⊥ is derived.
Set of formulas is saturated, no ⊥ derived.
A is valid.
There is an interpretation that falsifies A.

FIGURE 2.1: Summary of refutational theorem proving

2.3 The Superposition Calculus

Superposition is a refutationally complete proof system developed for first-order logic with equality. It has been developed in early 1990s, by Bachmair and Ganzinger [BG90] and Nieuwenhuis and Rubio [NR92].

In this thesis, the version of superposition used by E theorem prover [Sch02] will be described. Before we treat superposition in more detail, we need to establish some notation and define notions like substitution, most general unifier and term orders. Notation will closely resemble the one in the original E paper [Sch02].

Without loss of generality we assume all literals are equational (i.e. of the form $s \approx t$). If that is not the case, then we can introduce (fresh) symbol of type *bool*, **T**, and turn every atom $p(t_1, t_2, \dots, t_n)$ to $p(t_1, t_2, \dots, t_n) \approx \mathbf{T}$. Clauses are defined to be multisets of equational literals, where multiset is a mapping from a set S to set of integers. Intuitively, if N is a multiset then $N(x)$ is the number of copies of element x in N. We write $x \in M$ if $M(x) > 0$. Positive literals in a clause C are denoted as C^+, and will be written as $s \approx t$, whereas negative literals in a clause are written as C^- and referred to as $s \not\approx t$.

Definition 22. A substitution σ is a mapping from V_a to set of terms with type a, such that the set $\{x \mid \sigma(x) \neq x\}$ is finite. A substitution is called renaming if all terms in the codomain of σ are variables. Substitutions are lifted to terms and clauses in an intuitive manner (i.e. by mapping the variables that appear in a term or a clause).

Definition 23. A *ground simplification order* $>$ is a well-founded partial order on a set of terms that is stable under contexts and substitutions, total on variable-free terms and has subterm property.

In Definition 23, stability under context means that if $l > r$ then for any term t and position p, $t[p \leftarrow l] > t[p \leftarrow r]$. Similarly, stability (or closure) under substitution means that for any substitution σ and terms l, r if $l > r$ then $\sigma(l) > \sigma(r)$. Order $>$ has subterm property if $t > s$ for every subterm s of t, $s \not\equiv t$.

The order from Definition 23 is lifted to clauses using *multiset extension*. This extension is defined for any two finite multisets as follows:

Definition 24. Order $>$ is extended to finite multisets as an order $>_{mul}$ for which $N >_{mul} M$ if (1) $M \neq N$ and (2) for every x, if $M(x) >_{\mathbb{N}} N(x)$ then there exists $y, N(y) >_{\mathbb{N}} M(y)$ and $y > x$ ($>_{\mathbb{N}}$ is relation greater-than on the set of integers).

Definition 25. x is *maximal* with repsect to M if there is no element $y \in M$ such that $y > x$. x is *strictly maximal* with repsect to M if there is no element $y \in M$ such that $y \geq x$.

The way clauses are compared depends on how the equations are represented. Positive equations $s \approx t$ are represented as $\{\{s\}, \{t\}\}$, whereas negative equations $s \not\approx t$ are represented as $\{\{s, t\}\}$. Thus, in general, negative equation $s \not\approx t$ is greater than positive $s \approx t$.

Superposition calculus allows us to select clauses with which we can perform some inferences. With some constraints imposed on selection function, superposition calculus can block many unnecessary inferences.

Definition 26. A *selection function sel* is a function from clause C to multisubset of C with the property that $sel(C) \cap C^- = \emptyset \longrightarrow sel(C) = \emptyset$. In other words, *sel* has the property that if something is selected, then the selection has to contain at least one negative literal.

With those definitions in place, we can almost define the rules needed for the superposition calculus. Before that we need to define literals which can be involved in superposition inference rules and define the most general unifier.

Definition 27. Let $C = l \vee R$ be a clause and σ a substitution. We say $\sigma(l)$ is *eligible for resolution* if: (1) $sel(C) = \emptyset$ and $\sigma(l)$ is maximal in C or (2) $sel(C) \neq \emptyset$ and $\sigma(l)$ is maximal in $\sigma(sel(C) \cap C^-)$ or (3) $sel(C) \neq \emptyset$ and $\sigma(l)$ is maximal in $\sigma(sel(C) \cap C^+)$. Similarly, l is *eligible for paramodulation* if $l \in C^+, sel(C) = \emptyset$ and $\sigma(l)$ is maximal in $\sigma(C)$.

Definition 28. A *unifier* for terms s and t is a substitution σ such that $\sigma(s) = \sigma(t)$. The *most general unifier (mgu)* for s and t is a unifier with the property that any other unifier ρ for s and t can be expressed as $\rho = \sigma \circ \tau$ for some substitution τ, where $\circ$ is function composition.

Now we have all the ingredients we need to state the four rules that comprise superposition proof system:

Equality resolution *(ER)*

$$\frac{s \not\approx t \vee R}{\sigma(R)}$$

where σ is mgu of s and t and $\sigma(s \not\approx t)$ is eligible for resolution.

Equality factoring *(EF)*

$$\frac{s \approx t \vee u \approx v \vee R}{\sigma(t \not\approx v \vee u \approx v \vee R)}$$

where σ is mgu of s and u, $\sigma(s) \not< \sigma(t)$ and $\sigma(s \approx t)$ is eligible for paramodulation.

Superposition into negative literals *(SN)*

$$\frac{s \approx t \vee S \quad u \not\approx v \vee R}{\sigma(u[p \leftarrow t] \not\approx v \vee S \vee R)}$$

where σ is mgu of s and $u|_p$, $\sigma(s) \not< \sigma(t), \sigma(u) \not< \sigma(v), \sigma(s \approx t)$ is eligible for paramodulation, $\sigma(u \not\approx v)$ is eligible for resolution and $u|_p \notin V_a$ where a is the type of $u|_p$.

Superposition into positive literals *(SP)*

$$\frac{s \approx t \vee S \quad u \approx v \vee R}{\sigma(u[p \leftarrow t] \approx v \vee S \vee R)}$$

where σ is mgu of s and $u|_p$, $\sigma(s) \not< \sigma(t), \sigma(u) \not< \sigma(v), \sigma(s \approx t)$ is eligible for paramodulation, $\sigma(u \approx v)$ is eligible for resolution and $u|_p \notin V_a$ where a is the type of $u|_p$.

Theorem 3. The superposition proof calculus is sound.

Theorem 4. The set of clauses β that is saturated with respect to the superposition proof calculus contains $\perp$ if and only if β is unsatisfiable.

2.4 The E Theorem Prover

The E theorem prover [Sch02]; [Sch13b] is an automatic theorem prover for many-sorted FOL with equality. It is based on the superposition calculus and if it doesn't run out of CPU or memory resources, refutationally complete. Next to implementing core superposition, E puts strong emphasis on rewriting and on avoiding inferences that can not contribute to a proof.

For many years E has been a top-contender at The CADE ATP System Competition (CASC). At the last competition E finished third after two versions of Vampire in the "first-order theorems category" [Sut17b].

E is written in ANSI C in a highly portable manner and has been compiled on many platforms. It is largely independent of external libraries. For example, it has its own memory management subsystem with a mark-and-sweep garbage collector for terms and I/O subsystem.

It is a saturating theorem prover, which means that it will perform inferences until it either derives the empty clause (that is $\perp$) or any new inference is redundant (redundancy will be defined later on). Its saturation loop is based on a form of the *given-clause* algorithm.

In short, this algorithm divides the proof state in two sets of clauses – P (processed clauses) and U (unprocessed clauses). At the beginning, all clauses are in U and P is empty. Then, one by one, clauses are chosen from U based on some heuristics. Let C be the clause chosen in one run of the saturation loop. If C is the empty clause, that shows the unsatisfiability of the original set of clauses. Otherwise, E performs all generating inferences from clauses in $P \cup \{C\}$ and puts their conclusions D into U. If U is empty and no C can be chosen then the set of clauses is saturated (up to redundancy) which means the initial formula has an interpretation that falsifies it.

This kind of loop is one of the reasons E performs well in practice. E's author Stephan Schulz gives some properties of the loop that contribute to its performance [Sch02]:

- Clauses in U are truly passive. This means that they will not be part of an inference unless activated by being chosen in the saturation loop. The consequence of this property is that there is very little work to be done for clauses in U.
- Throughout the main loop the invariant that clauses in P are saturated is kept. Furthermore, all clauses in P are maximally simplified with respect to other clauses in P.
- Since P is maximally simplified and saturated, the choice of a new clause is very important. Thus, E has a set of advanced heuristics that pick a given clause. Moreover, the search procedure is is parametrized on a fine-grained level.
- Since only relatively small sets are involved in costly operations, the throughput of clauses in the main loop is high.

One of E's main strengths is fine-grained proof search control. E can choose clause selection schemes, perform different literal selections and work with different term orders. An automatic mode that analyses input problem and chooses all of these parameters is present and performs well in competitions.

2.5 Lambda-Free Higher-Order Logic

The main difference between the higher-order logic (HOL) and first-order logic is that in HOL quantification over functions and predicates is allowed. From the beginning of automated deduction systems, even though most effort has been put in dealing with FOL, attempts have been made to tackle higher-order logic [BK98].

The version of HOL that we will consider in this thesis is based on the simply typed λ-calculus. Benzmüller and Kohlhase [BK98] give pointers to an introduction to this calculus.

The main goal of our project is extending a real-world FOL theorem prover to a fragment of HOL. In our HOL fragment, we disallow λ-abstraction, for the reasons given in Chapter 1. Hence, the terms in λ-free higher-order logic (LFHOL) are similar to the terms in FOL.

Before giving definition of terms, we will assume that non-logical symbols are defined over the *signature* $S = (F, (T, rank))$. Signature defines the function symbols (F), T contains types (with two special, built-in types o and i), whereas $rank : F \to T$, declares function's type (and implicitly, maximal arity). Similarly, for each type A, we assume infinitely countable set of variables V_A.

Definition 29. Higher-order types are defined recursively as follows:

1. An atomic type a is a type.
2. If T_1 and T_2 are types then $T_1 \rightarrow T_2$ is a type.

Definition 30. LFHOL terms are defined recursively as follows:

1. Variable $X_A \in V_A$ is a LFHOL term of type A.
2. Constant $c \in F, rank(c) = A$ is a LFHOL term of type A.
3. If term s is an LFHOL term of type $A \rightarrow B$ and t is an LFHOL term of type A then $s\,t$ is a LFHOL term of type B.

We define set of *subterms* of term $t \equiv s_1\,s_2$ similar to the FOL case. The set of subterms of t contains t and all subterms of s_1 and s_2.

Example 2. Suppose we are given a function symbol f of type $a \rightarrow (b \rightarrow c)$, g of type $a \rightarrow b$, c_1 of type a and c_2 of type b. In this example and the rest of this thesis we will use a convention that variables are written in uppercase letters with their type as an index. Furthermore, for denoting types we will use symbol $\rightarrow$ as right-associative. Lastly, we will use term application as left-associative.

Having those function symbols declared and conventions in place, the following would be legal LFHOL terms:
(1) $f\,c_1\,c_2$, (2) $f\,c_1\,(g\,c_1)$, (3) $f\,c_1$, (4) g
(5) $X_{a\rightarrow b}\,c_1$, (6) $X_{a\rightarrow b}\,Y_a$, (7) $F_{(b\rightarrow c)\rightarrow d}\,(f\,X_a)$

FOL interpretation of terms and formulas can be gracefully extended to LFHOL. As a reference for HOL interpretation we used Andrew's *Introduction to Mathematical Logic and Type Theory* [And86].

An interpretation $I = (D, I_D, \nu)$ for signature $S = (F, (T, rank_F))$ is defined as follows. Let D be a family of disjoint non-empty sets indexed by higher order types A, whose elements are called *domains*. D_o has exactly two elements, **T** (true) and **F** (false).

Furthermore, let I_D be a mapping from function symbols to objects of the appropriate domain defined as follows: If $f \in F$, with $rank_F(f) = A$, then $I_D(f) = \mathbf{f}$, where **f** is an element of D_A. Type A might be complex (i.e. $A \equiv A_1 \rightarrow A_2$). In that case, object **f** can be applied to an object of domain D_{A_1}. Note that we do not separate function symbols from predicate symbols in LFHOL. However, we can consider as a predicate symbol any function symbol whose return type is o.

ν is a family of variable valuations indexed by higher order types A. Each ν_A maps set of variables V_A to domain D_A. With $\nu_A[x_A := c_A]$ we label the valuation that is exactly the same as ν_A with the only difference that mapping of variable x_A is forced to be $c_A \in D_A$. In the following, we will assume an interpretation I, for signature S.

Terms are interpreted similarly to FOL:

1. If X_A is a variable of type A, then $I_T(X_A) = \nu_A(X_A)$.
2. If c is a constant of type A, then $I_T(c) = I_D(c)$.
3. If t_1 and t_2 are terms and type of t_1 is $A \rightarrow B$ and type of t_2 is A, then $I_T(t_1\,t_2) = I_T(t_1)(I_T(t_2))$.

With (LF)HOL we have a choice of either fixing the logical connectives like we did in FOL or we can define them as functions that operate on the type o. We decided to keep the definitions as close as possible to FOL.

In LFHOL, we keep the same set of logical connectives as in FOL. Furthermore, we keep the same interpretation of these connectives – the logical formulas are interpreted as member of the domain D_o.

In HOL, predicates could be applied to other terms that are of type o. This can include logical formulas as well. In contrast, LFHOL disallows supplying logical formulas as arguments of predicates. More precisely, atoms in LFHOL are either terms of LFHOL that have return type o or equations $s \approx t$ where neither s nor t are of type o, but have the same type.

Note that the lack of lambda abstraction makes LFHOL strictly weaker than many other fragments of HOL. However, by adding appropriate axioms we can reclaim the power of HOL [Ker91]. Consider the following examples:

Example 3. Formula

$$(f\ x\ y \approx g\ y\ x) \longrightarrow (\exists H_{t\to t\to t}.\ H_{t\to t\to t}\ x\ y \approx g\ y\ x)$$

is provable in both standard HOL and LFHOL. In LFHOL, we can instantiate H with f and we would easily get the proof of the formula.

Example 4. Formula

$$\exists H_{t\to t\to t}\ .\ H_{t\to t\to t}\ x\ y \approx g\ y\ x$$

is provable in for standard HOL , but not for LFHOL. In HOL, instantiating H with $\lambda X_t\ Y_t.\ g\ Y_t\ X_t$ would lead us to the proof of formula. In LFHOL, there is no function symbol to instantiate H with, effectively deeming formula unprovable.

HOLs are also differentiated by the axioms they assume. Thus, an exposition of a HOL fragment (such as LFHOL) would not be complete if we don't list important formulas that are (in) valid in LFHOL.

Definition 31. *Axiom of Comprehension* (*AC*) states that for every formula A, there is a term such that it has the same value as the formula. In other words:

$$\exists u\ \forall v_1\ \dots\ \forall v_n\,.\,u\ v_1\ \dots\ v_n \approx A$$

where u does not appear in A and $v_1, \dots, v_n$ are distinct variables.

In HOLs that allow lambdas, we can introduce lambda abstraction (over the variables $v_1, \dots, v_n$) that returns formula A. This would be the witness for the existentially qualified u. Thus, for HOLs with lambdas AC is valid. However, LFHOL does not allow lambdas, which makes AC not valid.

Definition 32. *Axiom of Functional Extensionality* (*AFE*) states that functions whose results are equal for every argument are the same:

$$\forall f_{A\to B}\ \forall g_{A\to B}\ (\forall X_A\ \ .f\ X \approx g\ X) \longrightarrow f \approx g$$

In the models of LFHOL the elements of the set $D_{A\to B}$ are function-like objects, not necessarily (all) total functions from D_A to D_B. This means that AFE will not be valid in LFHOL. AFE can be stated in LFHOL and thus it can be made part of the input problem. It is a well known fact that, in practice, treatment of AE outside the proof calculus makes the search space grow fast. For that reason, we might implement a version of the superposition calculus that treats AE using special inference rules in the future.

Definition 33. *Axiom of Choice* (*ACh*) states that there is a function that chooses an element from every non-empty set:

$$\exists \xi_{(A \to o) \to A}\ \forall P_{A \to o}\ (\exists X_A\ .\, P\ X) \longrightarrow P\ (\xi\ P)$$

As a consequence of the fact that domains contain arbitrary function-like objects, ACh is not valid in LFHOL. We treat ACh in much more detail when we describe Skolemization in Section 6.1.

One can also look at LFHOL from a different perspective – using its isomorphism with the applicative FOL. More precisely, we can use the same set of connectives as in FOL and use a special encoding to translate LFHOL terms to FOL terms. In what follows, we are going to assume, without loss of generality, that all atoms are equational (i.e. of the form $s \approx t$).

We encode the LFHOL terms using *applicative encoding*. That is, for a given LFHOL problem with a signature S, we create FOL signature S_{FO}, that contains the same function symbols as S, but in addition, for each type $A \to B$ it contains a special binary application symbol $@^{A \to B}$. To define the translation function for LFHOL terms, we assume that there exists a bijective mapping $trans_\tau$ from the set of higher-order types T, corresponding to S, to the set of first-order types T_{FO}, corresponding to S_{FO}. That is, we assume that for each HO type $A \in T$ there is type $A_{\mathrm{FO}} \in T_{\mathrm{FO}}$, and vice versa. The bijective mapping *trans* from LFHOL terms to FOL terms is defined inductively as follows:

1. $trans(X_A) = X_{trans_\tau(A)}$, where X_A is a LFHOL variable of type A, or a LFHOL constant of type A, and $X_{trans_\tau(A)}$ the corresponding FOL variable or constant of type $trans_\tau(A)$.

2. $trans(t_1\ t_2) = @^{A \to B}(trans(t_1), trans(t_2))$, assuming that the type of t_1 is $A \to B$, and the type of t_2 is A.

Knowing that the set of logical connectives is the same for LFHOL and FOL, with the *trans* function, we have fully embedded LFHOL in FOL. Furthermore, for each FOL model of the translated LFHOL formula, we can construct a corresponding LFHOL model.

Let us show how FOL models can be converted to LFHOL models. In FOL models each $@^{A \to B}$ symbol is interpreted as a function from $D_{trans_\tau(A \to B)} \times D_{trans_\tau(A)}$ to $D_{trans_\tau(B)}$. In other words, when this function is given arguments $f \in D_{trans_\tau(A \to B)}$ and $a \in D_{trans_\tau(A)}$, it returns $b \in D_{trans_\tau(B)}$. Using this information, we can interpret LFHOL constants of complex types $A \to B$ as function-like objects whose return values correspond to the values returned by interpretation of $@^{A \to B}$.

Chapter 3

Generalizing Types and Terms

Terms are foundational data structure that take central place in many theorem prover architectures. Throughout the development of theorem provers many representations of terms have emerged with various benefits and drawbacks.

The main goal of this project is to extend the first-order theorem prover to a part of higher-order logic in a graceful manner. Making a smooth transition to LFHOL would be unimaginable if the term representation has to be changed. Thus, we decided to keep the term representation mostly the same, but extended it so that it supports HO features, without altering the behavior for FOL.

An important decision is how the theorem prover will deal with types. This decision is far-reaching as well, but one could argue that the changes to the type system have less impact since the types are not a central part of the theorem prover (some modern theorem provers do not even feature support for types). Our experience also showed that changes in types had much less *ripple effect* on other parts of the prover. The same is definitely not true for changes in terms.

3.1 Types

3.1.1 Types in the Original E

As of version 2.0 Turzum, E has support for many-sorted FOL[1], thanks to an update by Simon Cruanes. In what follows, we are going to use E's notation and refer to the types of constants (and simple types in general) as sorts.

Sorts are internally represented in the original E as an integer that corresponds to the type throughout the theorem proving process. Additionally, forward (sort to integer) and backward (integer to sort) indices are kept.

Function types are represented as a structure that has two important fields. The first field is an array that keeps sorts corresponding to arguments that function takes. The second field is the return sort of the function. Furthermore, in the signature, for each function and predicate symbol the type is stored. However, type checking is performed only during parsing since most of the inferences are type-preserving and in a few corner-cases type preservation is achieved using defensive programming.

Each term has a field that keeps the term's sort (`type`), so with each term only one integer has to be kept. This makes type comparisons easy and efficient – it amounts to comparing two integers.

[1]`http://wwwlehre.dhbw-stuttgart.de/~sschulz/WORK/E_DOWNLOAD/V_2.0/NEWS`

3.1.2 Types in hoE

This representation of types is not suitable for higher-order types. In FOL all terms are fully applied. This means that given a term t with a function symbol f at the root, the return sort of f will be the type of the term t. This is not true for LFHOL.

For example, consider a function symbol f of type $(a \rightarrow b) \rightarrow b \rightarrow c$. In LFHOL, the term $f\ X_{a \rightarrow b}$ is a valid LFHOL term of type $b \rightarrow c$. In the current implementation of E, f's type could not be represented, and the type of term would have to be encoded in an unnatural way for (LF)HOL. Thus, a representation that does not make a strict distinction between argument and return types is suitable.

Since E assumes that the types of terms could be compared in constant time (using C's == operator), this invariant had to be preserved in hoE as well. This is a constraint we kept in mind throughout designing the new type system.

Higher order types can be represented *natively*, the way they are described in Chapter 2 (i.e. as $\rightarrow\ (T_1, T_2)$, where T_1 or T_2 could themselves be complex) or in *flattened* representation (e.g. $\rightarrow\ (T_1, T_2, \dots, T_n)$ where all of the arguments except for T_n could be complex types). In other words, we use the fact that a type T can be uniquely decomposed in a number of arguments, with the last one always being a simple type.

For example, the type $(a \rightarrow b) \rightarrow c \rightarrow d$ can be represented in native representation as $\rightarrow\ (\rightarrow\ (a, b), \rightarrow\ (c, d))$ (where we make implicit assumption that $\rightarrow$ is right-associative, or in flattened representation as $\rightarrow\ (\rightarrow\ (a, b), c, d)$.

The representation that we chose for hoE is the flattened representation. The main reason is that checking for the type of ith argument that function f can take would be in $O(i)$ with native representation, whereas we could check that in $O(1)$ with the flattened representation.

Furthermore, we found it convenient to add support for types that can be encoded as sorts, but some users would like to write them more naturally. Namely, we added support for type constructors other than $\rightarrow$ (> in TPTP syntax). These type constructors can be defined using `thf type` syntax, with the constraint that all of the arguments are kinds (`$tType`). For example, if we have a type constructor `list` that takes one argument, we can define it in hoE as
`thf(listType, type, list: $tType > $tType).`

Nonetheless, polymorphism is not supported and those type constructors have to be provided with non-variable arguments, which justifies our claim that they could be encoded as sorts (e.g. `list_num` for `list(num)`).

Thus, to support higher order types and type constructors we generalized the type structure and defined it as

```
typedef struct typecell
{
   TypeConsCode      f_code;
   int               arity;
   struct typecell** args;
   TypeUniqueID      type_uid;
} TypeCell, *Type_p;
```

In this representation, the following invariants are kept:

1. The `f_code` field will have value 0 if and only if the type is constructed with $\rightarrow$
2. If the type constructor takes no arguments, its `args` field will have a value of `NULL`. Otherwise it will point to an array of exactly `arity` elements.

Lastly, we had to deal with the fact that previously types have been compared directly using C's == operator. However, as stated before, types of terms might not be simple sorts, so we need to establish some kind of sharing of types that have exactly the same arguments and use the same type constructor.

To that end we implemented type sharing, in exactly the same way E shares terms (see Section 3.2). Thus, after sharing the types that are structurally the same, we can again perform type comparisons efficiently using C's == operator (now comparing pointers, not integers).

One could notice that the type structure has fields that one would expect the term structure to have as well. Thus, it is natural to think that types can be represented using the facilities that E already has in place for the terms.

For the first few weeks of hoE development we tried exactly that approach. However, we observed many downsides. The most important one is that it introduces cyclic dependency in many modules that were nicely layered and largely independent before. For example, the signature module has to store types for each function symbol. If types are represented as terms, suddenly the signature would depend on types that would depend on the signature to provide function name to function code mapping. Thus, in terms of code elegance and coupling this solution proved unfruitful.

3.2 Terms

3.2.1 Terms in the Original E

One of the most consequential choices when implementing a theorem prover is the way in which the terms are represented. E represents terms as directed acyclic graphs (DAGs) in which repeated terms are guaranteed to be shared (*perfectly shared terms*). This means that the terms that are structurally the same will be the same object in memory, which has several important consequences.

First, sharing of subterms can save substantial amounts of memory. Experiments performed by Löchner and Schulz [LS01] are concerned with the *sharing factor*, which represents the ratio of the number of terms in the proof state and the number of unique terms in the proof state. Most of the problems exhibit the sharing factor that is between 5 and 100, which backs up the intuition that perfect sharing saves memory.

Second, rewriting of a shared term has as a beneficial side-effect of rewriting all the terms the shared term represents. By comparing a prover that does not use shared term representation (Waldmeister) to E, and setting their parameters to work in a very similar manner Löchner and Schulz observed that the number of rewrite operations is around the same in both provers for small problems and grows to around 2.5 times more rewrite operations for Waldmeister.

Last, perfectly shared terms allow for very efficient equality comparison. Namely, since perfectly shared terms keep the invariant that the terms that are structurally the same are the same object in memory, it is enough to test if the pointers to the term objects are the same to test for the term equality.

However, there is a penalty to be paid when using shared terms. Namely, when a new term is created the *term bank* is queried to check if the term is already present in the proof state. If so, the pointer to the already present term is returned. Otherwise, the query term is placed in the term bank.

The term bank is organized as a large hash table in which collisions are solved using closed addressing [Cor+09]. More precisely, an ad hoc total order on terms is established and using this order terms that have the same hash value are stored in a splay tree, as shown in Figure 3.1.

This organization has two benefits. On the first level, hashing should decrease the number of terms E compares by a huge constant (roughly the size of the table). On the second level, splay trees have an average $O(\log n)$ search (and insert) complexity and have $O(\log n)$ amortized complexity for the same operations. Furthermore, they favor objects that are accessed recently – for example, if a rewrite rule repeats many variables in the pattern possibly many queries for a subterm will be issued and answered in $O(1)$ term comparisons.

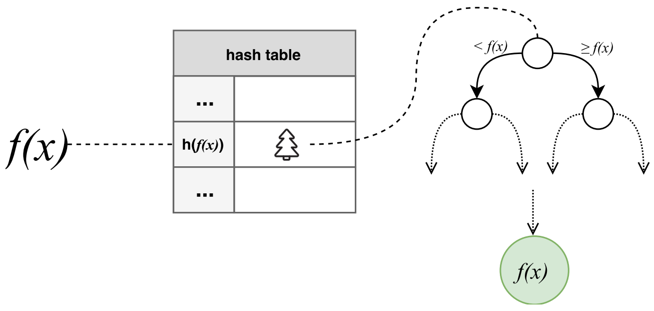

FIGURE 3.1: Term banks

The individual term cells in E are represented using a relatively complicated structure that has 11 fields at the moment. Due to the number of fields in the structure, we will describe them when needed.

The most important fields are `f_code`, which corresponds to the term's root function symbol; `arity`, which corresponds to the number of arguments; and `args`, an array that contains pointers to argument subterms. Note that the `arity` field is redundant (since arity of each function symbol is fixed in FOL and in E it is stored in the signature object), but it is stored in term object nevertheless.

3.2.2 Terms in hoE

In LFHOL, function symbols are not as tightly coupled to their arguments as it is the case for FOL terms. Namely, it can be seen in Definition 30 that the only way to construct complex terms is using application. However, LFHOL terms can be uniquely decomposed in $\psi\, s_1\, s_2\, \ldots\, s_n$, where ψ is either a variable or a function symbol (called *head*) and $s_1\, s_2\, \ldots\, s_n$ are arguments that might be applications themselves [Bec+17].

We decided to use this decomposition to represent the terms. An alternative option was to represent the terms using the applicative encoding. In other words, we could represent each term as an application of one subterm to the other. For example, the LFHOL term $f\,(X_{t_1 \to t_2}\, a)\, b$ would be represented as $@(@(f, @(X_{t_1 \to t_2}, a)), b)$ in the applicative encoding, where @ is the binary application function symbol.

The decision to represent terms decomposed (i.e. flattened) is tightly linked with the decision to use the same representation for types. Since we wanted the advantage of tight coupling of term head and arguments, we needed to facilitate efficient checking of ith argument's type. This would be considerably more complex with the non-flattened representation of types.

As explained before, the consequences of choosing a certain term representation are far-reaching. The main reason for using the decomposition term representation is that it is very close to native FOL term representation that E works with. For example, the heuristics that E is known for do not have to be reworked for LFHOL terms. Specifically, they will determine the head of the term just by reading the `f_code` field of the term. In other words, our extension of the term data structure is graceful – it keeps all of the useful properties it had for the FOL terms, while showing benefits for LFHOL. There are many other reasons for using the flattened representation, but we will explain them in the chapters that follow.

Having the previous description in mind, it is obvious that the term representation does not have to be changed at all for the terms that have a non-variable head. However, for terms which have a variable as head symbol, representation has to be changed. Namely, for the hoE representation of applied variables the `f_code` field has a value less than 0, but its `arity` field is non-zero and its `args` array holds the arguments. This breaks the original E invariant that variables have no arguments.

Breaking the invariant that variables have no arguments might seem as a small and an innocent change, but it has an expressed ripple-effect. In the original E, variables that appeared multiple times in the same clause are shared (i.e. they are the same object in memory). With applied variables represented as described above in hoE this sharing invariant would be broken.

Example 5. Consider the clause $f\ (X_{t\to t}\ a)\ Y_t \approx g\ Y_t\ X_{t\to t}$. The subterm $X_{t\to t}\ a$ would be represented as an object that has X's function code and a as an argument. On the other hand X on the right-hand side of the equation would be a separate object represented as a (usual, original E) variable.

The failure to preserve variable sharing invariant would have rather subtle effects. Namely, during matching and unification (see Chapter 4), variables are bound to terms. The term the variable is bound to is stored in the `binding` pointer. Once the variable is bound, it is of crucial importance that all occurrences of the same variable have the same binding.

With the sharing invariant in place, this will be ensured automatically. However, hoE breaks the sharing invariant since the variable at the head of the term will be disconnected from other occurrences of the variable in the clause (it will be a different object in memory).

One way to circumvent this problem is to iterate through the clause and fix the `binding` field manually. However, this is costly and very inelegant. Furthermore, it is not in line with E's spirit, since E carries the nickname of a *brainiac theorem prover* for its smart and elegant use of advanced data structures and algorithms [Sch02].

The solution that we settled for is that whenever we have a variable as the head of the term and that variable is applied to some arguments, we encode it as a term with a special function symbol called `$@_var` with constant function code and put the head variable as the first argument. In that way, variables can retain all of the properties they had in original E (including perfect sharing).

However, we kept in mind that `$@_var` is not a *real* function symbol and we have undone this encoding in many places where this would introduce problems (calculating the weight and the depth of the term, for example).

To conclude how we generalized terms and to better explain the memory layout of E, we will illustrate it in an object diagram. In Figure 3.2, the in-memory representation of the term $f\ (g\ a)\ (X_{t\to t\to t}\ (g\ a)\ b)\ Y_t$ is depicted.

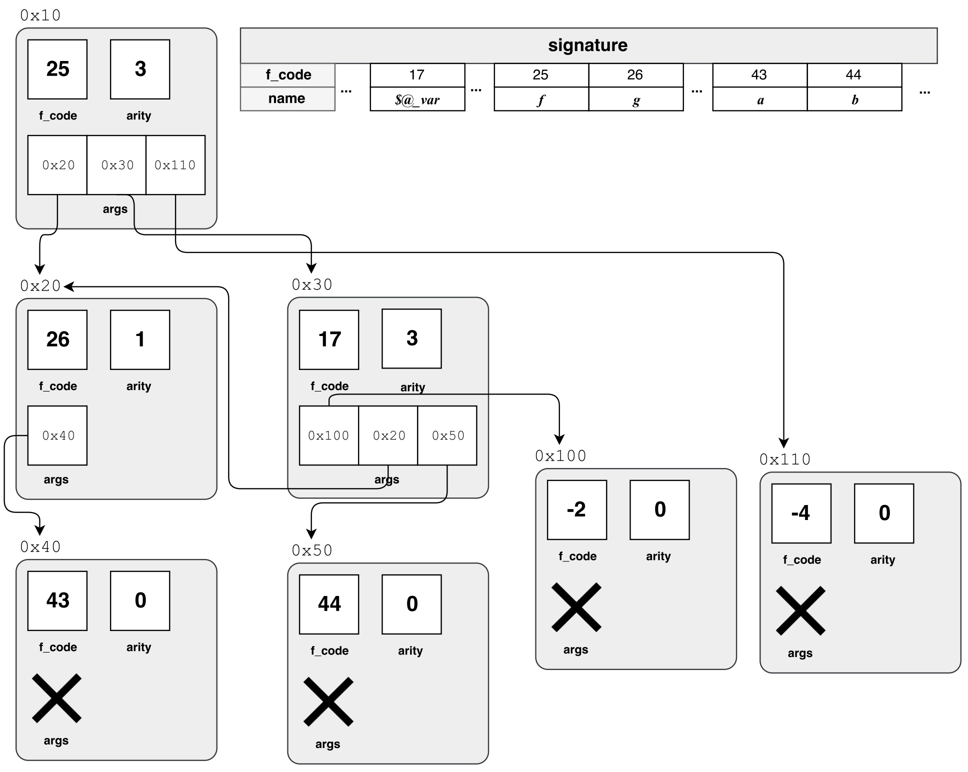

FIGURE 3.2: Memory layout of the term $f\ (g\ a)\ (X_{t\to t\to t}\ (g\ a)\ b)\ Y_t$

3.3 Knuth-Bendix Order

In Section 2.3, we stated that the superposition calculus needs a term order to determine literals that are eligible to be part of an inference rule. E supports the Knuth-Bendix order (KBO) and the lexicographical path order (LPO). Becker et al. have generalized KBO [Bec+17] to LFHOL terms and showed the benefits this generalization has over first-order KBO on applicative encoding of LFHOL terms. In hoE, we implemented this generalization of KBO, which will be described in what follows.

3.3.1 First-order Knuth-Bendix Order

The Knuth-Bendix order on first-order terms $>_{\mathrm{KBO-FO}}$ is parametrized with a partial order $\succ_F$ on the set of function symbols F and a weight function φ that assigns an integer (weight) to function symbols and variables. The weight function is extended to (non-variable and non-constant) terms by $\varphi(f(t_1, t_2, \ldots, t_n)) = \varphi(f) + \varphi(t_1) + \varphi(t_2) + \ldots + \varphi(t_n)$. Furthermore, for each function symbol or variable ξ, we denote the number of occurrences of ξ in s as $|s|_\xi$

Definition 34. Assuming partial order $\succ_F$ on set of function symbols F and weight function φ, $>_{\mathrm{KBO-FO}}$ on terms s and t is defined recursively as follows:

1. if $s \equiv f(s_1, s_2, \ldots, s_n)$ and $t \equiv g(t_1, t_2, \ldots, t_m)$ then $s >_{\mathrm{KBO-FO}} t$ if
   - $|s|_x \geq |t|_x$, for all variables x (var-check) and
     (a) $\varphi(s) > \varphi(t)$ or
     (b) $\varphi(s) = \varphi(t)$, $f \succ_F g$ or

(c) $\varphi(s) = \varphi(t)$, $f = g$ and there exists some i, such that $s_1 \equiv t_1, s_2 \equiv t_2, \ldots, s_{i-1} \equiv t_{i-1}$ and $s_i >_{\text{KBO-FO}} t_i$

2. if $s \equiv f(s_1, s_2, \ldots, s_n)$, $t \equiv x$, where x is a variable that appears in s, then $s >_{\text{KBO-FO}} t$.

If one would implement the first-order KBO naively following the definition, repeated computations will be performed – the var-check and the weight computation are performed for subterms possibly many times. This would incur $O(N^2)$ time complexity, where $N = |s| + |t|$ ($|\cdot|$ is the number of function symbols or variables in a term).

Löchner has devised an elegant way to compute the result of the KBO comparison in $O(N)$ time [Löc06]. The main idea presented in Löchner's work is avoiding repeated work for subterms by using the *tupling method*. In the tupling method, the results for independent subcomputations are calculated and returned as a tuple to the caller function.

Furthermore, even if the weight computation and var-check are performed efficiently, KBO might have to perform lexicographic comparison of the argument tuple. Löchner's KBO also computes the lexicographical comparison bottom-up, so no computation will have to be performed twice on any subterm.

In E, KBO comparison is implemented in a function `KBO6Compare`, which is located in file `ORDERINGS/cto_kbolin.c`. This implementation completely corresponds to a version of linear KBO from Löchner's paper.

3.3.2 Knuth-Bendix Order Extended to LFHOL Terms

Becker et al. [Bec+17] have gracefully extended KBO to LFHOL terms. Their KBO extension fully coincides with FOL KBO on FOL terms. We decided to implement this extension, using the Löchner's tupling method.

This extension uses another parameter to KBO, a mapping *ghd* that can be used to compare even more LFHOL terms. Namely, by extending the precedence on function symbols $\succ_F$ using *ghd*, we can compare terms that have variable as a head to other terms that have other symbols at the head.

In our current implementation of hoE, we give up this ability and if there is a variable at the head of a term t, we can compare it only to other terms s that have the same variable at the head. In future versions of hoE, we may reverse this decision and try to find efficient ways to compare terms that have variables at the head.

Moreover, the extension in Becker et al.'s work allows argument tuples of different function symbols to be compared in different ways (e.g. using length-lexicographic order or multiset order extension). In our implementation we support only length-lexicographic comparison of argument tuples for all function symbols.

Definition 35. Assuming a partial order $\succ_F$ on a set of function symbols F and a weight function φ, $>_{\text{KBO-HO}}$ on terms s and t is defined recursively as follows:

1. if $s \equiv \zeta_1\, s_1\, s_2\, \ldots\, s_n$ and $t \equiv \zeta_2\, t_1\, t_2\, , \ldots\, t_m$, where ζ_1 is the head symbol of s and ζ_2 is a head symbol of t, in the sense of unique decomposition discussed in Section 3.2.2 then $s >_{\text{KBO-HO}} t$ if

   - $|s|_X \geq |t|_X$, for all variables X (var-check) and

     (a) $\varphi(s) > \varphi(t)$ or

     (b) $\varphi(s) = \varphi(t)$, neither ζ_1 nor ζ_2 are variables and $\zeta_1 \succ_F \zeta_2$ or

   (c) $\varphi(s) = \varphi(t)$, $\xi_1 = \xi_2$, $n > m$ or $n = m$ and there is some i, such that $s_1 \equiv t_1, s_2 \equiv t_2, \ldots, s_{i-1} \equiv t_{i-1}$ and $s_i >_{\mathrm{KBO-HO}} t_i$

2. if $s \equiv \xi\, s_1\, s_2\, \ldots\, s_n$, $t \equiv x$, where x is a variable that appears in s, then $s >_{\mathrm{KBO-HO}} t$.

Even though $>_{\mathrm{KBO-HO}}$ could compare FOL terms, our goal was to keep E's performance the same on FOL terms. Since term comparisons comprise up to 35% of E's runtime [Löc06], we decided to extend Löchner's linear implementation of KBO with length-lexicographic comparison and treatment of applied variables in a separate function that will be called only when E is supplied with a higher-order problem.

Extending the linear implementation of KBO to LFHOL incurred immaterial changes to FOL implementation. Namely, we had to account for length-lexicographical comparison and for the fact that applied variables can be at the head of the term.

In hoE, KBO comparison is also implemented in the function `KBO6Compare`, which is located in the file `ORDERINGS/cto_kbolin.c`. However, depending on whether the problem is higher-order or first-order different low-level code will perform comparison.

Chapter 4

Generalizing Matching and Unification

Unification stands as one of the conditions in each of the core inference rules of the superposition calculus. It is an important algorithm in many fields of computer science that has been studied for many decades. Furthermore, it is a well known fact that for HOL with lambdas unifiability of terms is not even decidable.

We give a LFHOL unification algorithm that coincides with FOL unification for FOL terms. Furthermore, the version of the algorithm that we give in this chapter preserves the time complexity of FOL unification algorithm. In the worst case, time complexity of the unification algorithm is exponential.

On the other hand, matching can be seen as a special case of unification, but efficient treatment of matching in the form of a separate algorithm is preferred. We gracefully extend the FOL matching to LFHOL, preserving linear time complexity of the FOL algorithm.

4.1 Matching

Given two terms s and t, the matching problem is finding a substitution σ, such that $\sigma(s) = t$. If such a substitution exists, we will call t an *instance* of s, and we will call s *more general* than t. σ is called a *matching substitution*. Additionally, if σ exists, we say s can be matched onto t.

The matching problem is inherently oriented. In other words, the matching finds a substitution that instantiates variables only in term s, whereas term t is passive. To avoid confusion and to have a uniform way to refer to arguments of matching procedure, we will call term s *pattern* and term t *target*.

One could solve the problem of LFHOL matching using the isomorphism with applicatively encoded FOL terms. However, this would be inefficient since we would need to perform a two-way translation. Thus, we created an algorithm that works with native LFHOL terms.

4.1.1 First-Order Matching Algorithm

In the spirit of Terese [Bez+03], we will present the matching algorithm by trying to solve a seemingly more general problem – matching of a set of equations.

Let $S = \{s_1 = t_1, \ldots, s_n = t_n\}$ be a set of equations. The matching problem for the set of equations consists of finding the substitution σ such that for each s_i, $\sigma(s_i) = t_i$. Our initial problem now easily reduces to finding the solution for the set $\{s = t\}$.

One way to solve the matching problem for a set of equations is given in Algorithm 1. In Algorithm 1 we call an equation *solved* if it is of the form $x = t$, where x

Algorithm 1 Matching algorithm for FOL terms

```
1: procedure MATCH(EquationSet $S$)
2:     while $S$ is not solved do
3:         Pick an unsolved equation $s = t$ from $S$
4:         if $s \equiv f(s_1, \dots, s_n)$ and $t \equiv f(t_1, \dots, t_n)$ then
5:             $S \leftarrow (S \setminus \{s = t\}) \cup \{s_1 = t_1, \dots, s_n = t_n\}$
6:         else if $s \equiv f(s_1, \dots, s_n)$ and $t \equiv g(t_1, \dots, t_m)$ and $f \neq g$ then
7:             return MATCHFAILED
8:         else if $s \equiv f(s_1, \dots, s_n)$ and $t \equiv x$, where $x$ is a variable then
9:             return MATCHFAILED
10:        else if $s \equiv x$, where $x$ is a variable and $(s = t') \in S$, for term $t', t' \neq t$ then
11:            return MATCHFAILED
12:    return MATCHSUCCEEDED
```

is a variable and $x = t', t' \neq t$ does not appear in S. Similarly, a set S is solved, if all of its equations are solved.

At the end of Algorithm 1, if there is a matching substitution σ, it will be stored in S, as a set of solved equations $x_i = u_i$.

4.1.2 E Implementation of the Matching Algorithm

E implements Algorithm 1 in a C procedure called `SubstComputeMatch`, located in the file `TERMS/cte_match_mgu_1-1.c`. The algorithm's complexity is linear in the size of the pattern and the target and it implements a heuristic to return failure for terms that are obviously not matchable.

Let function φ_{STD} be the standard weight function that assigns variable weights of 1 and function symbols weight of 2. It is lifted to terms recursively by $\varphi_{\text{STD}}(f(t_1, \dots, t_n)) = \varphi_{\text{STD}}(f) + \varphi_{\text{STD}}(t_1) + \dots + \varphi_{\text{STD}}(t_n)$.

Theorem 5. Let s be a term and σ a substitution. Then $\varphi_{\text{STD}}(\sigma(s)) \geq \varphi_{\text{STD}}(s)$.

Theorem 6. Let s be a pattern and t a target in matching problem. If $\varphi_{\text{STD}}(s) > \varphi_{\text{STD}}(t)$ then there is no substitution σ such that $\sigma(s) = t$.

Theorem 6 is a corollary of Theorem 5 (proved by a simple induction on the term structure). Namely, if s can be matched onto $t, \sigma(s) = t$, for a substitution σ. Thus, $\varphi_{\text{STD}}(\sigma(s)) = \varphi_{\text{STD}}(t)$. However, if $\varphi_{\text{STD}}(s) > \varphi_{\text{STD}}(t)$, then for each σ, $\varphi_{\text{STD}}(\sigma(s)) > \varphi_{\text{STD}}(t)$ by Theorem 5 and transitivity of $\geq$. In the end, we get $\varphi_{\text{STD}}(\sigma(s)) > \varphi_{\text{STD}}(t)$, which contradicts the fact that s can be matched onto t.

For representing the set S, E uses a stack D. The left hand side and the right hand side of each equation in S are represented as two consecutive stack entries.

At the beginning of the algorithm, the pattern s and target t are pushed (in that order) to D. This corresponds to initializing the set S with $\{s = t\}$.

At the beginning of the algorithm's main loop two terms t_i and s_i are popped from D. Having in mind the order in which terms are pushed to the stack, the term s_i corresponds to the left-hand side of the corresponding equation $s_i = t_i$.

Then, all four checks from Algorithm 1 are performed. If the head symbols of s_i and t_i match then it is an invariant of FOL that they have the same number of arguments. Thus, the arguments of s_i and t_i can be pushed to D in pairs where s_i's argument is pushed first. If the head symbols differ, failure is reported. This check also covers the case where s_i is a complex term and t_i is a variable, because the head symbol of s_i would have a positive `f_code`, whereas t_i would have a negative `f_code`.

Furthermore, if s_i is a variable its `binding` pointer is checked. If the `binding` pointer is set, this means that this equation is solved and s_i shouldn't be bound to any other term. Thus, if s_i's `binding` pointer points to the term t'_i, set S would have solution if and only if $t_i = t'_i$ (this corresponds to last check in Algorithm 1). If the `binding` pointer of s_i is not set then $s_i = t_i$ is part of the substitution, so E sets the `binding` pointer accordingly and adds $s_i = t_i$ to the substitution object.

If the term s can be matched onto t, at the end of `SubstComputeMatch`, the `binding` fields of variables in s will have the values mandated by σ and they will be recorded in the substitution object.

To show how E matches terms, we will illustrate it with a Figure 4.1 that shows the matching of $s \equiv f(x, g(y))$ against $t \equiv f(h(a,b), g(c))$

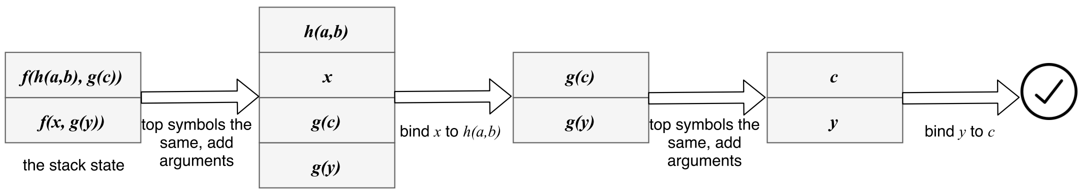

FIGURE 4.1: An example of matching FOL terms

4.1.3 Lambda-Free Higher-Order Matching Algorithm

LFHOL terms pose a few new challenges for term matching. The most important one is that LFHOL terms allow partial application and applied variables. Having this and the fact that the notion of subterms is quite different in LFHOL case in mind, it is not entirely clear if Algorithm 1 can be gracefully generalized. Using a few examples, we give an intuition about what kind of issues can appear when matching LFHOL terms and then generalize their implementation.

From the definition of subterms for LFHOL, it is obvious that even for a small term, the set of its subterms is relatively large. Namely, it can be shown that a LFHOL term $f\, t_1 \, \ldots \, t_n$ has almost twice as much subterms as the first-order term $f(t_1, \ldots, t_n)$ [BWW17]. In other words, if we interpret a LFHOL term as a FOL term, we would observe only about half the subterms a real LFHOL interpretation (with prefix subterms) would show.

Example 6. The set of subterms for LFHOL term $f\,(g\,a\,b)\,(h\,c)$ is $\{f\,(g\,a\,b)\,(h\,c), f\,(g\,a\,b), f, g\,a\,b, g\,a, g, a, b, h\,c, h, c\}$, whereas the set of subterms for FOL term $f(g(a,b), h(c))$ is $\{f(g(a,b), h(c)), g(a,b), a, b, h(c), c\}$.

For a given term s, E finds all subterms of a term $t \equiv f(t_1, \ldots, t_n)$ that s can be matched onto during some part of the proof search. To that end, E will first try to match s onto t and then perform the same operation recursively for subterms $t_1, \ldots, t_n$ (in some cases the order might be subterms-first).

However, LFHOL terms do not only have subterms that are arguments of the head symbol, but possibly also prefixes of the term. One obvious solution to facilitate the support for matching prefixes of terms is to change E's subterm traversal loop to explicitly create the prefix of the target and try to match the pattern onto it. This would not only be substantial change in terms of the amount of code to be added, but also wasteful. One could prove that at most one prefix of a term t can be matched by the term s. Thus, this approach would create an order of n prefix subterms and perform matching for each of those which is linear in the length of the term – incurring a substantial overhead.

Algorithm 2 Matching variable to a possible prefix of a term

```
1: procedure PARTIALMATCH(Term $var$, Term $t$, EquationSet $S$)
2:     Declare $args\_eaten$
3:     if SUFFIXEQUAL($var.type$, $t.type$) then
4:         $args\_eaten \leftarrow$ TYPEARROWARITY($t.type$) $-$ TYPEARROWARITY($var.type$)
5:     else
6:         return MATCHFAILED
7:     if $var$ is not bound in $S$ or $var$ is bound to $t[: args\_eaten]$ in $S$ then
8:         return $args\_eaten$
9:     else
10:        return MATCHFAILED
```

Example 7. Term $s \equiv f\,a\,b$ can be matched onto subterm $f\,a\,b$ of $t \equiv f\,a\,b\,c$. Note that there is no other prefix of t s can be matched onto.

Thus, we created a matching procedure that given a pattern s and a target t determines the only prefix of t that s can match onto. As a result, in the case of a successful matching, the amount of remaining arguments of t (*trailing arguments*) is returned.

Hence, the matching problem for LFHOL terms has to be stated slightly differently. To account for the fact that the target term can have trailing arguments, we can assume that we can concatenate some number of fresh (unused in any of the terms) variables to the pattern's argument tuple.

More precisely, the LFHOL matching problem is stated as follows: given terms s and t, where s has n arguments, and s's head symbol can be applied to at most k arguments ($k \geq n$), is there a substitution σ, and a tuple of fresh variables a of length $l \leq k - n$, such that $\sigma(s \cdot a) = t$, where $\cdot$ represents argument concatenation.

Matching prefixes is not only an optimization, however. It arises as a necessity when matching applied variables. In what follows, a k-length prefix of $t \equiv f\,s_1\,\ldots\,s_n$ ($k \leq n$) is denoted as $t[: k]$, where $t[: k] = f\,s_1\,\ldots\,s_k$. The prefix which consists of all arguments except for the last l ($l \leq n$) arguments is denoted as $t[: -l] = f\,s_1\,\ldots\,s_{n-l}$.

Example 8. Suppose that we try to match $X_{t \to t \to t}\,c\,d$ onto the term $f\,a\,b\,c\,d$. The substitution $\sigma = \{X \mapsto f\,a\,b\}$ would be a matching substitution (if $f\,a\,b$ has type $t \to t \to t$).

Theorem 7. Let $s \equiv X\,s_1\,\ldots\,s_n$ be a pattern and t be a target in the matching problem. Additionally, let X have type $T_1 \to \ldots \to T_n$, and the head symbol of t type $U_1 \to \ldots \to U_m$. Then s can be matched onto t if $m \geq n$ and $T_1 = U_{m-n+1}, \ldots, T_n = U_m$. Furthermore, in the matching substitution, X has to be bound to the prefix of length $m - n$ of the term t.

Calculation of the prefix subterm that variable X from Theorem 7 matches can be performed using Algorithm 2. This algorithm will be the crucial part of LFHOL matching procedure.

To the caller, LFHOL matching procedure returns the number of the trailing arguments of the target. On the higher level, the consequence is that E's subterm traversal schemes do not have to be changed at all to be able to take into account new subterms.

Even more importantly, this algorithm has exactly the same worst-case asymptotic complexity as the first-order matching algorithm. This is completely in line with E's *brainiac* spirit.

Algorithm 3 Matching algorithm for LFHOL terms

```
1: procedure FreshVarsNum(Term $pattern$, Term $target$)
2:     $missing\_pattern \leftarrow \textsc{MaxArity}(pattern.type) - pattern.arity$
3:     $missing\_target \leftarrow \textsc{MaxArity}(target.type) - target.arity$
4:     if $missing\_pattern - missing\_target \geq 0$ then
5:         return $missing\_pattern - missing\_target$
6:     else
7:         return MatchFail
8: procedure MatchLFHOLAux(EquationSet $S$)
9:     Declare $remaining\_args$
10:    while $S$ is not solved do
11:        Pick an unsolved equation $s = t$ from $S$
12:        if $s \equiv f\, s_1 \ldots s_n$ and $t \equiv f\, t_1 \ldots t_n$ then
13:            $S \leftarrow (S \setminus \{s = t\}) \cup \{s_1 = t_1, \ldots, s_n = t_n\}$
14:        else if $s \equiv f\, s_1 \ldots s_n$ and $t \equiv g\, t_1 \ldots t_m$ and $f \neq g$ then
15:            if $f$ is a variable then
16:                $p \leftarrow \textsc{PartialMatch}(f, t, S)$
17:                if $p \neq$ MatchFail and $n + p = m$ then
18:                    $S \leftarrow (S \setminus \{s = t\}) \cup \{f = t[: p], s_1 = t_{p+1}, \ldots, s_n = t_{p+n}\}$
19:                else
20:                    return MatchFail
21:            else
22:                return MatchFail
23:        else if $s \equiv f\, s_1 \ldots s_n$ and $t \equiv X$, where $X$ is a variable then
24:            return MatchFail
25:    return MatchSuccess
26: procedure MatchLFHOL(Term $pattern$, Term $target$)
27:    $trailing \leftarrow \textsc{FreshVarsNum}(pattern, target)$
28:    if $trailing \neq$ MatchFail then
29:        Create a copy of $pattern$, $pattern'$
30:        Append a tuple of fresh variables of length $trailing$ to $pattern'$
31:        return ($\textsc{MatchLFHOLAux}(\{pattern' = target\}), trailing$)
32:    else
33:        return (MatchFail, 0)
```

Lastly, the pseudocode for the LFHOL matching algorithm is given in Algorithm 3. Note that the check that a variable is not bound to multiple terms (i.e. a substitution maps variable to two different terms) now degenerates to a special case of the second check performed in Algorithm 3 (when $n = 0$).

4.1.4 hoE Implementation of LFHOL Matching Algorithm

hoE implements the Algorithm 3 in a C procedure called `SubstComputeMatchHO`, located in the file `TERMS/cte_match_mgu_1-1.c`. The procedure's signature is different from `SubstComputeMatchHO` because it returns not only a boolean, but a negative constant `NOT_MATCHED` if the matching failed, or the number of trailing arguments in the target if matching succeeded.

The main difference between Algorithm 1 and Algorithm 3 is that the latter tries to match not just one instance term, but possibly a prefix of it. This means that some arguments of the target term might be left as remainder.

At early phase of the development of hoE, we wanted to be able to print more debug information and have information about arguments remaining in the instance term. Note that in Algorithm 3, the remaining (trailing) arguments of t are ignored except for noting their number.

Thus, we represented the set S with two stacks D_s and D_t corresponding to the pattern s and the target t. When the term's heads match or applied variable gets matched to the prefix of term, we push the remaining arguments of a term to the corresponding stack. Note that if one would push arguments from right to left, then popping the term s_i from D_s and the term t_i from D_t would give a valid equation $s_i = t_i$.

After thorough testing of the LFHOL matching procedure, we have enough assurance that the procedure is bug-free and in future versions we will combine two stacks in one (just like it is done in E).

One more difference from the original E matching implementation is that making sure a variable is not bound to two terms is now performed with a more complex `TermIsPrefix` check (not just simple pointer equality comparison of `binding` pointer), since applied variables can match prefixes of terms, as shown in Example 9.

Example 9. Suppose we match the pattern $s \equiv X_{t \to t}\,(X_{t \to t}\, b)$ against the target $t \equiv f\,a\,b\,(f\,a\,b\,b)$. Clearly, $\sigma = \{X \mapsto f\,a\,b\}$ is a matching substitution. After the first step of the LFHOL matching algorithm, $X_{t \to t}$ would be bound to $f\,a\,b$. Then $X_{t \to t}\, b$ would be matched against $f\,a\,b\,b$. If we would now do the simple X's `binding` pointer comparison against the right-hand side (like it is done in the FOL case) of the equation we would get a mismatch. Instead (like in Algorithm 3) we should compare what a variable is bound to to the prefix of the right-hand side.

This extension of the first-order algorithm is graceful. This means that if the LFHOL algorithm would be given first-order terms it would give exactly the same result as FOL algorithm, with the same worst-case algorithm complexity. However, since E is highly optimized for FOL, we decided to run the old algorithm whenever E is ran on a FOL problem. The reason for that is that the applied variable checks and typing checks are redundant in this case and would add some overhead compared to the native FOL algorithm.

4.2 Unification

Given two terms s and t, the unification problem is finding a substitution σ, such that $\sigma(s) = \sigma(t)$. If such a substitution exists, we will call terms s and t *unifiable* and σ *unifier*.

Unlike matching, unification is symmetric. In other words, unifier of terms s and t exists if and only if unifier of terms t and s exists.

The existence of the most general unifier for LFHOL terms is a consequence of the isomorphism with applicative FOL encoding. Similarly to matching, the most general unifier could be found by leveraging this isomorphism. Since this approach is inefficient, we created an algorithm that deals with LFHOL terms represented natively.

Algorithm 4 Unification algorithm for FOL terms

```
1: procedure UNIFY($S$)
2:     while $S$ is not solved do
3:         Pick a violating equation $s = t$ from $S$
4:         if $s \equiv f(s_1, \dots, s_n)$ and $t \equiv f(t_1, \dots, t_n)$ then
5:             $S \leftarrow (S \setminus \{s = t\}) \cup \{s_1 = t_1, \dots, s_n = t_n\}$
6:         else if $s \equiv f(s_1, \dots, s_n)$ and $t \equiv g(t_1, \dots, t_m)$ and $f \neq g$ then
7:             return UNIFICATIONFAILED
8:         else if $s \equiv x$ and $t \equiv x$, where $x$ is a variable then
9:             $S \leftarrow S \setminus \{s = t\}$
10:        else if $s \equiv f(s_1, \dots, s_n)$ and $t \equiv x$, where $x$ is a variable then
11:            $S \leftarrow (S \setminus \{s = t\}) \cup \{t = s\}$
12:        else if $s \equiv x$, where $x$ is a variable and $x$ occurs in t then
13:            return UNIFICATIONFAILED
14:        else if $s \equiv x$, where $x$ is a variable then
15:            $S \leftarrow \{x = t\} \cup [x \mapsto t](S \setminus \{s = t\})$
16:    return UNIFICATIONSUCCEEDED
```

4.2.1 First-order Unification Algorithm

As in the case of matching, following Terese [Bez+03], we will solve a seemingly more general problem – unification of a set of equations S. Before we delve into giving algorithm for solving the unification problem we define some notation. First, a substitution σ is applied to a set of equations $S = \{s_1 = t_1, \dots, s_n = t_n\}$ by $\sigma(S) = \{\sigma(s_1) = \sigma(t_1), \dots, \sigma(s_n) = \sigma(t_n)\}$. Second, we will denote a substitution that maps a single variable x to a term t by $[x \mapsto t]$.

The unification algorithm follows roughly the same scheme as the matching algorithm. It deconstructs terms that have the same head symbols, solves the problem for the arguments, and binds variables to corresponding terms. However, unification is undirected, which means that a variable subterm in the right-hand side of the equation can match a more complex subterm of the left-hand side of the equation.

Unlike a matching substitution, the unifier σ will be applied to both sides of the equation. This is why in the unifier σ, no variable x can be bound to a term containing x. The check that this case does not happen is called *occurs-check*. This check can be costly and some unification algorithms (e.g. the one in Prolog) do not perform it, at the cost of unsoundness.

Example 10. Suppose we want to unify terms $s \equiv x$ and $t \equiv f(x)$. If we would apply substitution $\sigma = \{x \mapsto f(x)\}$ to s and t we would get terms $\sigma(s) \equiv f(x)$ and $\sigma(t) \equiv f(f(x))$. It is clear that since $\sigma(s) \neq \sigma(t)$, σ is not unifier (and moreover, it can be shown that no unifier exists for those two terms). Note that term s can be matched onto t.

Lastly, when a variable is bound in the unification algorithm, this binding is applied to the whole remaining set S. The need for this stems from the non-orientation of unification, since it is necessary to apply the effects of variable binding to both the left- and the right-hand sides of equations simultaneously.

We present pseudocode for the FOL unification algorithm in Algorithm 4. A set of equations S is called solved if each equation is of the form $\{x_1 = u_1, \dots, x_n = u_n\}$, where x_i does not appear in any right-hand side u_j of the equation in set S. Any equation that causes S not to be solved is called a *violating* equation.

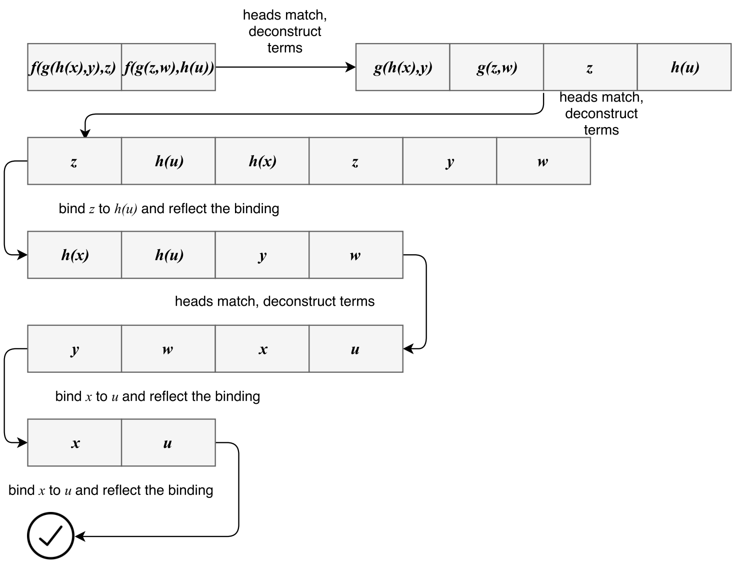
heads match, deconstruct terms
f(g(h(x),y),z)
f(g(z,w),h(u))
g(h(x),y)
g(z,w)
z
h(u)
heads match, deconstruct terms
z
h(u)
h(x)
z
y
w
bind z to h(u) and reflect the binding
h(x)
h(u)
y
w
heads match, deconstruct terms
y
w
x
u
bind x to u and reflect the binding
x
u
bind x to u and reflect the binding

FIGURE 4.2: Unification of first order terms

4.2.2 E Implementation of the Unification Algorithm

hoE implements Algorithm 4 in a C procedure called `SubstComputeMgu`, located in the file `TERMS/cte_match_mgu_1-1.c`. The worst-case time complexity of this procedure is exponential. Furthermore, the result of Theorem 6 is no longer applicable to unification, so `SubstComputeMgu` cannot use weight-optimization.

E implements the set S as a queue where consecutive elements correspond to the right-hand side and the left-hand side of the equation. The reason why E uses a queue instead of a stack in the case of unification is that E tries to delay the application of the rule in line 15 of the Algorithm 4 as much as possible. For that reason, equations of form $x = t$, where $t \not\equiv x$ and x is a variable are put to the back of the queue and considered last.

Namely, in practice, most of the tries to unify terms will fail, usually because the top symbols of the equations do not match. Finding this mismatch as early as possible would speed up the unification procedure.

Variables act as wildcards in unification procedure. In other words, they can match any term of the same type. Unifying a variable would delay finding possible mismatches. This would have even further-reaching consequences since a bound variable would have to be applied to the whole set of equations – incurring another slowdown (though E handles substitution application rather efficiently). Thus, E uses delaying binding variables as a heuristic that enables it to find mismatches faster.

Lastly, we illustrate the process of unifying terms $s \equiv f(g(h(x), y), z)$ and $t \equiv f(g(z, w), h(x))$ in Figure 4.2.

4.2.3 Lambda-Free Higher-Order Unification Algorithm

Most of the extensions that are related to matching prefixes and applied variables carry over to the unification algorithm. However, the algorithm's inherent lack of orientation posed new problems for our prefix match optimization.

Example 11. The term $f\ a\ X_t$ is unifiable with a subterm $f\ Y_t\ b$ of a term $f\ Y_t\ b\ c$. Similarly, the subterm $g\ c$ of a term $g\ c\ d$ is unifiable with a term $Z_{t \to t \to t}\ c$.

As one can see in Example 11, arguments can remain in either of the parameters to the unification procedure. Thus, the LFHOL unification procedure does not only have to determine if a term unified with a prefix, but check possibility of either side unifying with a prefix of the other side.

Like the matching problem, the unification problem has to be stated differently in LFHOL. Given terms $s \equiv f\, s_1\, \ldots\, s_n$ and $t \equiv g\, t_1\, \ldots\, t_m$, where f can be applied to at most k arguments ($k \geq n$), g to at most l arguments ($l \geq m$) unification problem is restated for LFHOL as follows: is there a substitution σ, and a tuple of fresh variables a of length $i \leq k - n$, such that $\sigma(s \cdot a) = \sigma(t)$ or a substitution ρ and a tuple of fresh variables b of length $j \leq l - m$, such that $\rho(s) = \rho(t \cdot b)$, where $\cdot$ is argument tuple appending operation.

Furthermore, line 11 of Algorithm 4 reorients an equation in which the right-hand side is a variable. LFHOL unification has to generalize this reorientation since it might happen that both on the left-hand side and the right-hand side a variable appears as a head symbol, and only one orientation of the variable binding leads to a unifier.

Example 12. The term $s \equiv Y_{t \to t \to t \to t}\, a\, a\, b$ and the term $t \equiv X_{t \to t \to t}\, a\, b$ are unifiable, with the unifier $\sigma = \{X \mapsto Y\, a\}$.

In Example 12, to determine the correct unifier the equation has to be oriented as $t = s$. Using the results of Theorem 7 we always reorient equations in a way that an applied variable that has lower maximal arity is the left-hand side of the equation. If the maximal arities are the same, the orientation is arbitrary.

The complete pseudocode for the LFHOL unification procedure is given in Algorithm 5. We modified PARTIALMATCH to PARTIALUNIFY to disable checks that are only applicable to the matching algorithm. FRESHVARSUNIFNUM calculates the exact number of fresh variables that have to be added to one side for unification to potentially succeed.

In conclusion, even though the unification algorithm might be considerably more complex, we managed to preserve several properties that are of major importance to this project. Firstly, we achieve complete compatibility with FOL unification. This means that the extension is graceful. Secondly, this algorithm incurs only constant-time overhead when unifying FOL terms. And lastly, it determines the maximal prefix matched, which means that no additional subterms other than FOL subterms will have to be traversed to perform all of the inferences E performs.

4.2.4 hoE Implementation of LFHOL Unification Algorithm

hoE implements the Algorithm 5 in a C procedure called `SubstComputeMguHO`, located in the file `TERMS/cte_match_mgu_1-1.c`. This procedure is exponential in the sum of the size of the arguments. It also does not return only boolean as the result, but a `UnificationResult` object that plays the role of the tuple in Algorithm 5.

Similar to the LFHOL matching algorithm, we used two stacks to store equations. Due to the fact that relatively serious complications have arisen with applied variables, in this version of hoE we did not implement E's optimization of delaying binding variables. In the first hoE update, we will unify two stacks and consider the changes needed to delay binding variables.

In the hoE implementation we manged to combine the procedures PARTIALMATCH and PARTIALUNIFY to achieve even further code sharing. Like matching, LFHOL unification is ran only when the E's input problem is higher-order.

Algorithm 5 Unification algorithm for LFHOL terms

```
procedure FRESHVARSUNIFNUM(Term $s$, Term $t$)
    $missing\_s \leftarrow$ MAXARITY($s\,.\,type$) $-$ $s\,.\,arity$
    $missing\_t \leftarrow$ MAXARITY($t\,.\,type$) $-$ $t\,.\,arity$
    return $missing\_s - missing\_t$
procedure PARTIALUNIFY(Term $var$, Term $t$)
    Declare $args\_eaten$
    if SUFFIXEQUAL($var.type$, $t.type$) then
        $args\_eaten \leftarrow$ TYPEARROWARITY($t.type$) $-$ TYPEARROWARITY($var.type$)
    else
        return UNIFICATIONFAIL
    return $args\_eaten$
procedure UNIFYLFHOLAUX(EquationSet $S$)
    while $S$ is not solved do
        Pick a violating equation $s = t$ from $S$
        if $s \equiv \varphi\, s_1 \ldots s_n$ and $t \equiv \varphi\, t_1 \ldots t_n$ then
            $S \leftarrow (S \setminus \{s = t\}) \cup \{s_1 = t_1, s_2 = t_2, \ldots, s_n = t_n\}$
        else if $s \equiv \varphi\, s_1 \ldots s_n$ and $t \equiv \gamma\, t_1 \ldots t_m$ and $\varphi \neq \gamma$ then
            if neither $\varphi$ nor $\gamma$ are variables then
                return UNIFICATIONFAILED
            else if both $\gamma$ and $\varphi$ are variables and $n > m$
                or $\gamma$ is a variable and $\varphi$ is not then
                $S \leftarrow (S \setminus \{s = t\}) \cup \{t = s\}$
            else if $\varphi$ is a variable then
                $p \leftarrow$ PARTIALUNIFY($\varphi, t, S$)
                if $p \neq$ UNIFICATIONFAIL and $n + p = m$ then
                    $S \leftarrow (S \setminus \{s = t\}) \cup \{\varphi = t[: p], s_1 = t_{p+1}, \ldots, s_n = t_{p+n}\}$
                else
                    return UNIFICATIONFAIL
        else if $s \equiv X$ and $t \equiv X$, where $X$ is a variable then
            $S \leftarrow S \setminus \{s = t\}$
        else if $s \equiv X$, where $X$ is a variable and $X$ occurs in t then
            return UNIFICATIONFAILED
        else if $s \equiv X$, where $X$ is a variable then
            $S \leftarrow \{X = t\} \cup [X \mapsto t](S \setminus \{s = t\})$
    return UNIFICATIONSUCCESS
procedure UNIFYLFHOL(Term $s$, Term $t$)
    $trailing \leftarrow$ FRESHVARSUNIFNUM($pattern, target$)
    Declare $trailing\_side$
    Create a copy of $t$, $t'$ and a copy of $s$, $s'$
    if $trailing < 0$ then
        Append a tuple of fresh vars of length $|trailing|$ to $t'$
        $trailing\_side \leftarrow$ TRAILINGLEFT
    else
        Append a tuple of fresh vars of length $trailing$ to $s'$
        $trailing\_side \leftarrow$ TRAILINGRIGHT
    if UNIFYLFHOLAUX($\{s' = t'\}$) = UNIFICATIONSUCCESS then
        return $(trailing\_side, trailing)$
    else
        return UNIFICATIONFAIL
```

Chapter 5

Generalizing the Indexing Data Structures

During the proof search a theorem prover has to consider a large number of clauses as an inference partner to a given clause. The superposition calculus restricts the possible number of inference partners by using a complex set of constraints for each inference rule.

The task of the indexing data structures is to filter only those clauses that might be eligible as an inference partner to a given clause. The main reason for their use is that by avoiding the naive search through the proof state, we can save a substantial amount of time and increase the clause throughput. If the indexing data structure returns exactly those clauses that are eligible for the inference we call that indexing structure *perfect*.

One recurring constraint in all of the inference rules of the superposition calculus is that a term in one clause must be unifiable with a term in another clause. Furthermore, as we will see in Chapter 6, there are other constraints put on terms in inference and simplification rules. Some of the operations one needs from an indexing data structure are

- **Finding unifiers:** For a query term t, find all terms s such that s and t are *unifiable* ($\sigma(s) = \sigma(t)$, for some substitution σ).
- **Finding instances:** For a query term t, return all *instances* s of t ($s = \sigma(t)$, for some substitution σ).
- **Finding generalizations:** For a query term t, return all *generalizations* s of t ($\sigma(s) = t$, for some substitution σ).

One could conclude from this list that indexing techniques work only on (sets of) terms. However, there are structures that index clauses. In other words, those structures filter only clauses that satisfy some constraint. Even though we cover *subsumption* in detail in Chapter 6, for the sake of completeness, we give a list of operations one needs from an index working on a clause level:

- **Finding subsumers:** For a query clause C, find all clauses D such that $\sigma(D) \subseteq C$ for some substitution σ.
- **Finding subsumed clauses:** For a query clause C, find all clauses D such that $\sigma(C) \subseteq D$ for some substitution σ.

5.1 Perfect Discrimination Trees

5.1.1 Description of the Data Structure

The problem of searching digital text for patterns is one of the fundamental problems in computer science. Because of its importance it has been researched thoroughly and many efficient algorithms that solve this problem have been devised.

One way to represent the text as a set of words which supports various querying operations is to organize words in a tree structure called *trie*.[1] In a trie, characters of the word are used to find a path to the node that stores some information related to the word (e.g. the number of occurrences). In other words, each node has a set of edges that are labeled with characters of the alphabet [Knu98]. The root of the tree has edges that are labeled with the first characters of the indexed words, the nodes on the second level are labeled with the second characters of the indexed words, and so on. Tries can perform a lot of operations on a set of words efficiently – checking if the word is present, counting occurrences or finding words with common prefix, to name a few. Tries can be used to index not only strings, but tuples of any objects that have a way to be represented as a string.

In FOL, each function symbol is always passed the same number of arguments. This means that if we know the arity of function symbols, we can unambiguously represent the term as a string, omitting all of the parentheses and commas. This string is exactly the string one would get by printing each term symbol in preorder traversal (depth-first, left-to-right).

Example 13. A term $t \equiv f(g(c), h(g(x), g(f(z, y))))$ can be represented as $f\ g\ c\ h\ g\ x\ g\ f\ z\ y$. Similarly, if we were given a string $f\ g\ c\ h\ g\ x\ g\ f\ z\ y$ and the information about arities of symbols, we could reconstruct the term $f(g(c), h(g(x), g(f(z, y))))$ assuming the arities are as in t.

Now that we have the means to represent the terms flattened as strings, we can reuse data structures (such as tries) that are used to index strings to employ them in indexing terms.

One of the data structures that are reusing tries is the *Discrimination Tree*. This data structure can work in two modes – as a perfect or imperfect index. The imperfect version is easier to implement but has worse performance in practice. The nodes in this tree are labeled with function symbols or variables and each path from the root to a leaf in this tree corresponds to one preorder traversal flattening of a term. Since all of the FOL terms have unique flattening, each leaf corresponds to one and only one term.

Discrimination trees are usually presented in their imperfect form in literature [McC92]; [RV01]. This means that all the variables are replaced with the wildcard symbol that matches any term, and the retrieval operations will return a superset of all the terms that satisfy the query condition. By contrast, E implements the perfect form of discrimination trees called *Perfect Discrimination Trees* (PDTs).

E uses PDTs only for finding *generalizations*. For that reason, we are going to describe only this algorithm in what follows. We will present the algorithm in the spirit of the *Handbook of Automated Reasoning* [RV01], generalizing it to many-sorted FOL and the perfect form of indexing.

We first need to introduce some notation. The position of the subterm that will be visited right after $t|_p$ for some position p and term t in preorder traversal is called

[1]Pronounced as ”try”, stemming from re*trie*val.

Algorithm 6 Finding generalizations in a PDT

```
1: procedure GENERALIZATIONSAUX(TreeNode $n$, Term $t$, Position $p$)
2:     $Res \leftarrow \emptyset$
3:     if $n$ is a leaf then
4:         $Res \leftarrow$ Terms stored in $n$
5:     else
6:         if HEAD($t|_p$) is a function symbol adjacent to $n$ then
7:             $neighbor \leftarrow$ node corresponding to HEAD($t|_p$), adjacent to $n$
8:             $Res \leftarrow$ GENERALIZATIONSAUX($neighbor, t$, NEXT($t, p$))
9:         for all variables $x_a$ of the same type as $t|_p$ that are adjacent to $n$ do
10:            $neighbor \leftarrow$ node corresponding to $x_a$, adjacent to $n$
11:            $bound \leftarrow False$
12:            if $x_a$ is not bound to any term then
13:                $bound \leftarrow True$
14:                Bind $x_a$ to $t|_p$
15:            if $x_a$ is bound to term $t|_p$ then
16:                $Res \leftarrow Res \cup$ GENERALIZATIONSAUX($neighbor, t$, AFTER($t, p$))
17:            if $bound$ then
18:                Unbind $x_a$
19: procedure GENERALIZATIONS(PDTIndex $i$, Term $query$)
20:     return GENERALIZATIONSAUX($i.root$, $query$, $\varepsilon$)
```

next(t, p). The position of the subterm that will be visited right after all subterms of $t|_p$ have been visited is called *after*(t, p).

The pseudocode for finding generalizations is given in Algorithm 6. This algorithm matches the query term t by following all possible paths in a PDT that might generalize the query term. At some position p during the search for generalizations, if the head symbol of the term $t|_p$ matches some of the adjacent nodes' label, we can follow this link since we have observed no violation of matching compatibility so far and move to the next symbol of the preorder traversal of t. Similarly, if there is an adjacent node that is labeled with an unbound variable that matches $t|_p$'s type, we can bind the variable to $t|_p$ and advance to the position p' that corresponds to the symbol observed only after all subterms of $t|_p$ have been traversed, and follow the path to the node that corresponds to the variable.

However, there are a few cases where we can't find a generalization from the current PDT node. If there is no adjacent node that matches the head symbol of the query term, there is no substitution that can make the term stored in the PDT match the query term. Similarly, if there is no variable of the matching type or the variable is already bound to another term, we can end that part of the search.

Figure 5.1 shows the representation of a PDT containing a set of terms $\{f(x, a), f(x, i(y)), f(g(x), h(y)), f(g(c), h(d)), g(x)\}$. Arrows show how this tree is traversed when finding generalizations of $f(g(a), i(a))$. A blue arrow means that the matching succeeded, whereas a red arrow means that the matching failed. The numbers above the arrow indicate the order in which the subtrees are traversed.

Algorithm 6 returns the generalizations present in the PDT index. However, for some use cases it might be more convenient to iterate through the resulting terms one at a time. The way in which results are returned is called *mode* of retrieval [RV01]. Storing the traversal state explicitly (e.g. using a stack), instead of using recursion,

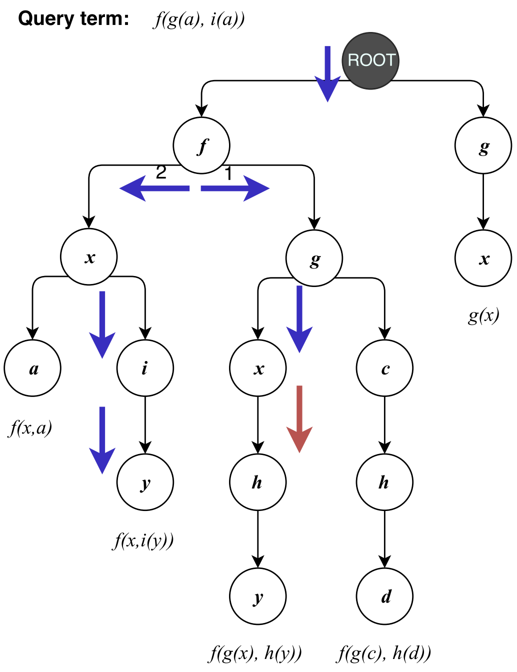

FIGURE 5.1: Finding a generalization in PDT

makes it possible to implement one-at-a-time retrieval mode.

5.1.2 E's Implementation of PDT

E implements an optimized one-at-a-time PDT retrieval mode. It implements a two-tier architecture where the user observes a high-level index structure, which itself contains the root of the PDT in which the actual matching happens. The whole PDT implementation is located in the file `CLAUSES/ccl_pdtrees.c`

The high-level data structure keeps most of the information related to the state of the search and the PDT nodes keep the information if we already matched function symbol and how many variables we have considered so far.

PDTs are used as an iterator – the user is expected to first call `PDTreeSearchInit`, after which each call to `PDTreeFindNextDemodulator` will return generalizations, until it returns `NULL`. The user is then expected to free the resources used for the search by calling `PDTreeSearchExit`.

Each PDT node has two integer maps: `f_alternatives` and `v_alternatives`. The first map maps function symbols to the successor nodes, while the second map maps variables to the successor nodes.

The query term is flattened on demand. This means that E keeps the stack D_{args} on which it flattens the term as matching happens in the PDT. More precisely, the abstract *next* operation of Algorithm 6 corresponds to pushing all the arguments of the current query term right-to-left to D_{args}. In this way the top of the stack always stores the term that is to be visited in the preorder traversal. In contrast, the abstract operation *after* corresponds to popping the term from the D_{args}.

After following some path of the PDT it can happen that the matching will fail on this path. PDT traversal algorithm then backtracks to the first position where the choice is possible. But backtracking does not only mean that we move upwards in the PDT, it also means that we consider a different position in the query term – that is, we have to undo the *next* or *after* operation.

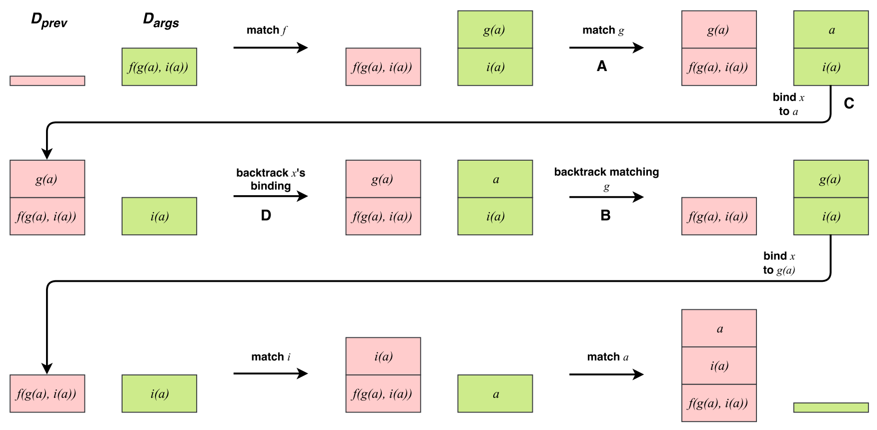

FIGURE 5.2: E's stack management while finding generalizations of $f(g(a), i(a))$

Undoing the following of the PDT node labeled with a function symbol corresponds to moving up the tree and undoing the *next* operation. When we followed the edge, we performed *next* by pushing the arguments $t_n, \ldots, t_1$ of the term $t|_p$ to D_{args}. An example of this operation is labeled with **A** on Figure 5.2 which shows how E finds generalization of term $f(g(a), i(a))$ in the tree from Figure 5.1.

When we have to undo this operation, we follow the edge in the reverse direction. By reading the edge's label f we could reconstruct the previous state, by popping n terms $t_1, \ldots, t_n$ from D_{args} where n is f's arity. However, this newly constructed term, though structurally equivalent to the one from the previous search state, is not the same object in memory (it is not shared yet). To avoid sharing the term (i.e. inserting it in the term bank) E stores all of the terms that we deconstructed by matching their top symbol on the D_{prev} stack and instead of constructing the term again, just uses the one from the top of D_{prev}. This is labeled with **B** on Figure 5.2.

When we have to backtrack matching the variable against a term, we have to move up the tree and backtrack the operation *after*. The term the variable is bound to is stored in the `binding` field of the term object, and *after* corresponds to binding the variable to the top of D_{args} and popping the top afterwards (**C** on Figure 5.2)

This operation is easier to backtrack, as label **D** from Figure 5.2 shows. It is enough to push the term the variable is bound to back to D_{args} and move up in the tree.

As an optimization, E uses Theorem 6 to further reduce the number of nodes it traverses during the search for generalizations. In each node n, E stores the standard weight of term that has the maximal standard weight among all the terms reachable from n. If the query term has the weight higher than this stored value, we can backtrack right away.

E supports two traversal orders of the PDT. If we flipped the order in which function symbol and variable nodes are visited, we would get the same results in the end observed as a set. But the order in which we would get them would be different. If we would first traverse variables, we would get more general terms first.

5.1.3 E Bug

While working on LFHOL extension of PDTs, we have discovered a serious bug in E. The presence of the bug was confirmed by E's author, Stephan Schulz. Fixing the bug is planned for the near future.

Namely, E has been built around the assumption that each distinct variable (with its unique `f_code`) has only one type. However, with a recent update by Simon Cruanes that introduced type support, this invariant has been broken. As a result, E 2.0 allows two variables (which are two different objects in memory) with the same `f_code` to have different types.

The consequences of breaking this uniqueness invariant are that if E stores multiple variables with the same `f_code`, but different types in the PDT, they are going to overwrite each other after each insertion of the variable to the PDT. Only the last added variable is going to be bound, but all of the variables stored in this node will be returned one by one.

This bug manifests itself only if the variables are the root of the PDT. Namely, if the variables are below the PDT root then their type is mandated by the position in the term that they occupy. This means that below the root, variable can have one and only one type, so different variable objects will have different `f_code`s.

For this bug to be triggered, the input (first-order) problem needs to have multiple clauses of the form $x \approx t_i$. Our guess is that the bug has not been discovered so far since it only influences simplification engine and is unlikely to interfere with soundness and completeness of the prover. However, we have not investigated the effects of the bug on first-order problems fully. Thus, we cannot be entirely sure if, in presence of this bug, E keeps both soundness and completeness.

5.1.4 LFHOL Extension of PDTs

One of the main underlying assumptions for PDTs was that each function symbol is supplied a constant number of arguments. This obviously does not hold for LFHOL terms.

Example 14. Consider the terms $s \equiv f\,(g\,b)$ and $t \equiv f\,g\,b$. These two terms are both flattened to $f\,g\,b$. Thus, without further information, the flattening $f\,g\,b$ cannot be unambiguously converted back.

However, all terms of LFHOL are typed. Using the typing information, we can unambigously convert back and forth the flattened string and the usual representation of terms. For example, at most one of the terms s and t from Example 14 can be correctly typed. Towards a contradiction, suppose both terms can be correctly typed. Since f is applied, it must have type $A \to B$ (it cannot be atom type a). Since it is applied to g directly, g has to have type A for t to be correctly typed. Using the same argument A can be deconstructed as $A_1 \to A_2$ and b must be of type A_1. On the other hand, for s to be correctly typed $g\ b$ has to have type A. Since g is applied to b in s, g's type is of the form $A_1 \to A$. However, in our type system no finite type A can be defined as $A := A_1 \to A$.

Even though the flattened string unambiguously represents one and only one LFHOL term, there are still fundamental differences between standard FOL PDTs and our extension of PDTs to LFHOL. In FOL PDT, since every function symbol is applied to the same number of arguments, no string corresponding to a flattening of a term can be a prefix of another term's flattening. This means that a node that stores a term necessarily needed to be a leaf.

Example 15. The flattening of the term $h\,(g\,b)$ is $h\,g\,b$. On the other hand, flattening of the $h\,(g\,b)\,a$ is $h\,g\,b\,a$ (assuming h's arity is larger than 2). The node that stores $h\,(g\,b)$ in LFHOL PDT will have an adjacent node storing $h\,(g\,b)\,a$.

Algorithm 7 Finding generalizations in a LFHOL PDT

```
 1: procedure ArgNum(Term $var$, Term $t$)
 2:     if SuffixEqual($var.type$, $t.type$) then
 3:         return $\textsc{TypeArrowArity}(t.type) - \textsc{TypeArrowArity}(var.type)$
 4:     else
 5:         return MatchFailed
 6: procedure GenAuxHO(TreeNode $n$, Term $t$, Position $p$)
 7:     $Res \leftarrow \emptyset$
 8:     if $n$ contains terms then
 9:         for all terms $t$ contained in $n$ do
10:             $Res \leftarrow Res \cup \{(t, \textsc{RemainingArgs}(t, p))\}$
11:     if $\textsc{Head}(t|_p)$ is a function symbol adjacent to $n$ then
12:         $neighbour \leftarrow$ node corresponding to $\textsc{Head}(t|_p)$, adjacent to $n$
13:         $Res \leftarrow Res \cup \textsc{GenAuxHO}(neighbor, t, \textsc{Next}(t, p))$
14:     for all variables $x_a$ that are adjacent to $n$,
15:         for which $\textsc{ArgNum}(x_a, t|_p) \neq \textsc{MatchFailed}$ do
16:         $pref\_len \leftarrow \textsc{ArgNum}(x_a, t|_p)$
17:         $neighbor \leftarrow$ node corresponding to $x_a$, adjacent to $n$
18:         $bound \leftarrow False$
19:         if $x_a$ is not bound to any term then
20:             $bound \leftarrow True$
21:             Bind $x_a$ to $pref\_len$-length prefix of $t|_p$
22:         if $x_a$ is bound to $pref\_len$-length prefix of $t|_p$ then
23:             $Res \leftarrow Res \cup \textsc{GenAuxHO}(neighbor, t, \textsc{After}_{\textsc{HO}}(t, p, pref\_len))$
24:         if $bound$ then
25:             Unbind $x_a$
26: procedure GeneralizationsHO(PDTIndex $i$, Term $query$)
27:     return GenAuxHO($i.root$, $query$, $\varepsilon$)
```

Thus, one needs to make sure that the during the traversal, terms are not returned only from leaves but from possibly any intermediate node.

Furthermore, we need to generalize the *after* operation, because in the LFHOL case a variable might not match the whole subterm of a query term, but possibly a prefix. The $after(t, p)$ operation needs to take this into account and move the position p to the first position after the matched prefix. This generalization of *after* depends on the length of the matched prefix, which we have to supply as the argument. We will denote the *after*'s generalization as $after_{\text{HO}}$.

The need for this change stems from the special form of higher-order term context that is not present in FOL. Namely, by equality congruence property from equation $h = g$ one can conclude $h\,X_t = g\,X_t$ and $h\,X_t\,Y_t = g\,X_t\,Y_t$ if h's type is $t \to t \to t$.

Lastly, like matching (and for the same reasons), we want PDTs to find generalizations that possibly match the prefix subterm of the query term. For that reason, the PDT will not only find a generalization, but return the number of arguments trailing in the query term. This can be fully determined from the position in which we end the matching.

Example 16. Term $f\,X_t\,Y_t$ is the generalization of the $f\,(g\,a)\,(g\,b)$ subterm of $f\,(g\,a)\,(g\,b)\,c$.

This generalization of FOL PDTs for LFHOL is entirely graceful. If all the terms stored in the PDT are FOL terms, querying for generalizations would return the same terms as before (in the same order). Moreover, the only place in the algorihtm where time complexity might increase is matching of the prefix of the term (line 16). Using a simple heuristic that would check if the variable's type is the same as the type of term at position p, we can conclude that we have to match the whole term. This means that we can bind the term in $O(1)$ again. Thus, for FOL terms the time complexity remains the same as well.

5.1.5 Implementation of LFHOL PDTs in hoE

hoE implements an optimized one-at-a-time PDT retrieval mode. No changes to the architecture of the PDT module have been performed, only small changes that enable all of the higher-order features of Algorithm 7. The whole PDT implementation is located in the file `CLAUSES/ccl_pdtrees.c`.

One of the fundamental differences between FOL and LFHOL PDTs is that terms can be stored in the inner nodes. However, as for the implementation this did not require a lot of changes in the code. The most notable change is that every node is queried for terms that it might store and that in a few places in term insertion and deletion procedure changes were made to lift assumptions that only leaves can store terms.

Prefix matching procedure from the unification and matching algorithms has been used again to match prefixes of the term in the case a (applied) variable appears in PDT. Furthermore, the same architecture with the two stacks used to traverse the term is used in LFHOL implementation of PDTs, with some specializations needed for applied variables and prefix matching.

Namely, when we match the prefix of the term, $\mathit{after}_{\mathrm{HO}}$ should move the position p right after the last prefix element. This corresponds to pushing the arguments of the term whose prefix is matched right-to-left to stack D_{args}, until we reach the argument that is element of prefix.

Example 17. Suppose that a variable $X_{t \to t \to t}$ has to be matched against $f\ a\ b\ c$ (where f has type $t \to t \to t \to t$). Then, it would be bound to $f\ a$ and the arguments c and b would be pushed to D_{args} (in that order). On top of D_{args} is the argument b, which is exactly the one that should be visited in preorder traversal after prefix matching. In other words, the operation of pushing the arguments to D_{args} in described manner corresponds to moving the term position using $\mathit{after}_{\mathrm{HO}}$.

The slight complication arises when we have to backtrack the choice to bind an applied variable. Namely, the prefix of the term would be in `binding`, and the rest of the term would be in some number of arguments on stack D_{args}. Without knowing how many arguments the original term had, there is no way to determine how many arguments we need to take from the stack D_{args} to undo the binding.

Example 18. Continuing the previous example, suppose that the variable $X_{t \to t \to t}$ has to be matched against $f\ a\ b$ (where f has the same type as before). Then, it would be bound to $f\ a$ and only the argument b would be pushed to D_{args}. Since $X_{t \to t \to t}$ is bound to exactly the same term as in the previous example, there is no obvious choice how many arguments to take from D_{args} to get the original term back.

We solved this problem by reusing the stack D_{prev} to hold the original term applied variable was matched against. This way, we can recalculate how many arguments we pushed to D_{args} initially and bring back both stacks to the original state – the state before binding the variable.

The bug that we briefly described in Section 5.1.3 has a major impact on hoE. In FOL variables appeared at the top of the PDT only if we had clauses of type $x \approx t_i$. Arguably, these clauses are fairly rare and would appear when one would like to describe a type that has only one inhabitant, for example. In LFHOL, the variable can appear as the head symbol, and incurs issues not only in the clauses of the previously described shape, but also whenever applied variable appears as the subterm.

Example 19. Suppose that f has the type $t \to t \to t$. Furthermore, suppose that both $t \equiv f\ a\ X_t$ and $s \equiv f\ a\ (X_{t \to t}\ a)$ are stored in the PDT. The term t is flattened to $f\ a\ X_t$ and s is flattened to $f\ a\ X_{t \to t}\ a$. When storing t in the PDT, the node corresponding to X_t is going to be created. When s is stored, this node will be rewritten with the one corresponding to $X_{t \to t}$, as an effect of the E bug.

Because of the prefix match, we additionally created the structure `MatchInfo` to enclose the information about the clause in which matched terms appear (that was the object returned previously) with the number of arguments trailing from the query term.

Lastly, generalizing this data structure was the most time-consuming change of the entire project. We had to thoroughly investigate the architecture of PDTs and find ways to use the existing architecture, while supporting HO operations. In that sense, this extension of LFHOL is entirely graceful. It maintains the same architecture and all the properties it had for FOL terms (e.g. the time complexity and the order in which terms are returned). The LFHOL extension also supports prefix matching as an important feature, but currently supports only shorter-first prefix match order. In the future versions of hoE we will try to make the prefix match order a choice, in the same way the generality order of the returned terms is a choice in current E implementation. This can be useful for term traversal schemes that are not only left-most innermost (see Chapter 6).

5.2 Fingerprint Indices

5.2.1 Description of the Data Structure

PDTs use the flattening of a term that can be arbitrarily long as the key. This may leave a heavy memory footprint if many long terms are stored in the proof state. Furthermore, implementing operations such as finding unifiable terms can be hard if efficiency is a consideration. Fingerprint indices try to solve this problem by having a fixed-size key for each term, with easy to implement query operations. However, fingerprint indices lose the perfect index property – they just return a superset of the terms that satisfy the query property.

The basic idea behind fingerprint indexing is that when a substitution is applied to a term, it can only add new function symbols to a term. Furthermore, any position that was valid in a term before applying substitution is valid after applying it [Sch12]. More precisely, considering a term t and a position p, only four cases are possible:

1. $t|_p$ is a variable;
2. $t|_p$ is a non-variable term;
3. p is not a valid position in t, but for some substitution σ, p is a valid position in $\sigma(t)$;

4. p is not a valid position in t and no substitution σ can make it a valid position.

Example 20. Let $t \equiv f(a, h(y))$. Then, $t|_{2.1}$ is an example of case 1. Terms $t|_1$ and $t|_2$ exhibit case 2. Position 2.1.1.2 is an example for case 3. Lastly, $t|_3$ and $t|_{1.2}$ are examples for case 4.

The above case distinction is useful for determining if terms are matching or unification compatible. If for a given position p and a term t, the head symbol of $t|_p$ is a function symbol f_1, for some other term s, p might not even be a valid position of $\sigma(s)$, for any substitution σ. This tells us that those two terms cannot be unified or matched (in any direction). Thus, for a carefully chosen set of positions one could efficiently determine if any violation of unification or matching compatibility conditions is observed.

Definition 36. Let $\Sigma = (F, P, T)$ be a signature and let $\mathbf{A}, \mathbf{B}$ and $\mathbf{N}$ be distinguished symbols not appearing in $F \cup P$. Let $\Phi = F \cup P \cup \{\mathbf{A}, \mathbf{B}, \mathbf{N}\}$. A generic fingerprint function *gff* that maps terms to the set Φ, based on a position p is defined as follows:

$$gff(t,p) = \begin{cases} \mathbf{A} & \text{if } t|_p \text{ is a variable} \\ f & \text{if the head of } t|_p \text{ is the function symbol } f \\ \mathbf{B} & \text{if } p = p_1.p_2\text{, for some } p_1 \text{ and } p_2 \neq \varepsilon \text{ and } t|_{p_1} \text{ is a variable} \\ \mathbf{N} & \text{otherwise} \end{cases}$$

Based on a fixed tuple of positions $\overline{p} = (p_1, \ldots, p_n)$, we define the fingerprint function $fp(t) = (gff(t, p_1), \ldots, gff(t, p_n))$. $fp(t)$ is called the *fingerprint*.

Intuitively, for a given position p, *gff* samples the property of term t that is exhibited on a position p and that could give us useful information about matching or unification. Similarly, *fp* gives a tuple of such properties increasing the amount of information that we can use to find possible matching and unification failures.

Using the formal definition of fingerprints we can speed up unification and matching. Assume that we want to sample the position p in terms s and t. Term s is compatible for matching onto term t if the value in intersection of row indexed by $gff(s, p)$ and column indexed by $gff(t, p)$ is marked with ✓in Table 5.1. Similarly, if in the intersection of row and column (indexed the same way) of Table 5.2 reads ✓, s is unification compatible with t. We lift the definition of matching and unification compatibility to fingerprints componentwise.

		target				
		f_1	f_2	**A**	**B**	**N**
pattern	f_1	✓				
	f_2		✓			
	A	✓	✓	✓		
	B	✓	✓	✓	✓	✓
	N					✓

TABLE 5.1: Matching compatibility matrix

	f_1	f_2	**A**	**B**	**N**
f_1	✓		✓	✓	
f_2		✓	✓	✓	
A	✓	✓	✓	✓	
B	✓	✓	✓	✓	✓
N				✓	✓

TABLE 5.2: Unification compatibility matrix

The key property of the fingerprints is that if s's and t's fingerprints are not matching-compatible then s cannot be matched onto t. Similarly, if s and t have fingerprints that are not unification-compatible, then they are not unifiable.

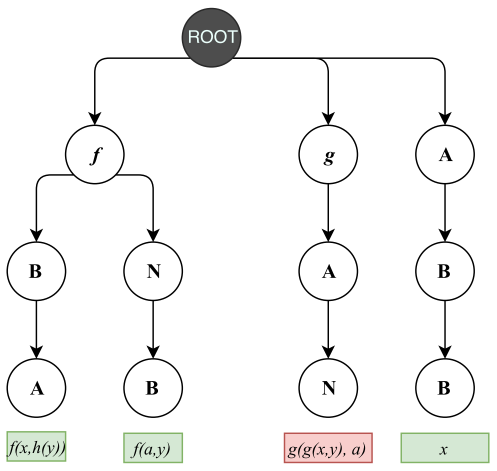

FIGURE 5.3: An example of a fingerprint index

Algorithm 8 Finding generalizations in a PDT

```
 1: procedure FINDFP(FPNode $n$, FPVector $fp$, Num $component$, SearchMode $mode$)
 2:     $Res \leftarrow \emptyset$
 3:     if $n$ is a leaf then
 4:         $Res \leftarrow$ Terms stored in $n$
 5:     else
 6:         for all nodes $next$ adjacent to $n$ and
 7:             for which $\textsc{Compatible}(next\,.\,label, fp[component], mode)$ do
 8:             $Res \leftarrow Res \cup \textsc{FindFP}(n, fp, component + 1, mode)$
 9:     return $Res$
10: procedure UNIFICATIONFP(FPIndex $i$, Term $query$)
11:     return FINDFP($i\,.\,root$, $query$, 1, UNIFICATION)
12: procedure GENERALIZATIONFP(FPIndex $i$, Term $query$)
13:     return FINDFP($i\,.\,root$, $query$, 1, GENERALIZATION)
14: procedure INSTANCEFP(FPIndex $i$, Term $query$)
15:     return FINDFP($i\,.\,root$, $query$, 1, INSTANCE)
```

We can use fingerprints directly as a heuristic. Before calling the matching or unification procedure, we can compare fingerprints componentwise and report failure if any component of the fingerprints is not compatible. By organizing fingerprint tuples in a trie-structure, fingerprints can be used in a one-against-many manner to return a superset of terms in the proof state that are matching or unification compatible.

Terms that can be matched or unified are found by following the paths in the fingerprint trie that are compatible. Since each term has a unique fingerprint, each indexed term will be stored at exactly one node. A single node can store multiple terms.

Algorithm 8 shows how fingerprint indices find terms that are compatible for a given operation. Fingerprinting allows for very modular code, since the core trie traversal procedure is the same except for using different compatibility criterion. The function COMPATIBLE queries the appropriate table based on the *mode* – that is, the query condition. The UNIFICATION mode finds unification compatible terms, which corresponds to querying Table 5.2. The GENERALIZATION mode finds terms that can be matched onto the query term, which corresponds to using the query term as a column index into Table 5.1. The INSTANCE mode finds terms that the query term can be matched onto, which means that the query term should be used as the row index into Table 5.1.

Figure 5.3 shows an example of a fingerprint index constructed with the position tuple $(\varepsilon, 1.1, 2.1)$. This index stores terms $\{f(x, h(y)), g(g(x,y), a), x, f(a,y)\}$. With green boxes we labeled the terms that are unification compatible with $f(b, h(z))$, whereas with red boxes we labeled the terms that are not unification compatible.

5.2.2 E Implementation of Fingerprint Indexing

Fingerprint tries are implemented in E in the file `TERMS/cte_fp_index.c` and the functions that obtain term's fingerprint in `TERMS/cte_idx_fp.c`.

E predefines 16 position tuples for constructing the fingerprint trie. Different position tuples offer different tradeoffs between the time needed to calculate the fingerprint and maintain the fingerprint index and the amount of matching and unification failures that can be observed. Schulz gives detailed experimental results for several position tuples [Sch12].

The operations supported by E's implementation of fingerprint indexing are finding unification compatible terms and finding terms that are possible instances of a given term. In Section 5.2.3, we describe in which parts of the proof search each operation is used in detail.

Operations are implemented using separate procedures for each operation. They closely follow the Algorithm 8, with inlining the COMPATIBLE check.

5.2.3 Extension of Fingerprint Indexing for LFHOL

The fundamental case distinction related to the term positions from Section 5.2.1 carries over to the LFHOL terms. However, with our prefix matching and unification optimizations in place, we must be careful how to use the information to filter out the terms that are not compatible.

Example 21. Suppose we sample position 1. Then for the term $s \equiv X_{t \to t \to t}\, c\, d$, $\mathit{gff}(s, 1) = c$, whereas for the term $t \equiv f\, a\, b\, c\, d$, $\mathit{gff}(s, 1) = a$. According to Table 5.1 and Table 5.2, their samples are neither matching, nor unification compatible. Yet, $\{X \mapsto f\, a\, b\}$ is a unifier (and a matching substitution from s onto t).

Thus, applied variables call for an altered generic fingerprint function. Coming with Example 21, we can see that the difference stems from the fact that for some substitution σ, $\sigma(s)|_1$ can be any term, not only c as FOL *gff* function would report. This fully coincides with the semantics of case 3 from Section 5.2.1.

Another type of issue emerges with prefix matching and unification optimizations in place. Term s from Example 22 matched onto (and unified with) a prefix of t, but FOL fingerprint compatibility matrix failed to capture this matching (unification).

Example 22. Suppose we sample position 1. Then for the term $s \equiv f$, $\mathrm{gff}(s, 1) = \mathbf{N}$, whereas for the term $t \equiv f\, a$, $\mathrm{gff}(t, 1) = a$. According to Table 5.1 and Table 5.2, their samples are neither matching, nor unification compatible. However, term identity is a unifier of s and $t[:0]$, as well as matching substitution from s to $t[:0]$.

The key observation is that in the matching (and unification) problem in LFHOL we allow a tuple of fresh variables of a certain (maximum) size to be appended to the arguments tuple. The *gff* function has to be altered to capture this as well.

Definition 37. We extend the FOL *gff* function to LFHOL $\mathit{gff}_{\mathrm{HO}}$ as follows:

$$\mathit{gff}_{\mathrm{HO}}(t,p) = \begin{cases} \mathit{gff}_{\mathrm{HO}}(t \cdot \overline{x}, p) & \text{if } t \text{ is not fully applied, and } \overline{x} \text{ is a tuple} \\ & \text{of fresh variables that makes it fully applied} \\ \mathbf{A} & \text{if } t|_p \text{ is a variable} \\ f & \text{if the head of } t|_p \text{ is a function symbol } f \\ \mathbf{B} & \text{if } p = p_1.p_2 \text{ where the head of } t|_{p_1} \text{ is a} \\ & \text{variable for some } p_1 \text{ and } p_2 \neq \varepsilon \\ \mathbf{N} & \text{otherwise} \end{cases}$$

Based on a fixed tuple of positions $\overline{p} = (p_1, \dots, p_n)$, we extend the fingerprint function $\mathit{fp}(t)$ to $\mathit{fp}_{\mathrm{HO}}(t) = (\mathit{gff}_{\mathrm{HO}}(t,p_1), \dots, \mathit{gff}_{\mathrm{HO}}(t,p_n))$. $\mathit{fp}_{\mathrm{HO}}(t)$ is called *LFHOL fingerprint*.

Example 23. Returning to Example 21, the LFHOL fingerprint sample of position 1 for the term s is **A**, whereas it stays a for t. The $\mathit{gff}_{\mathrm{HO}}$ function now reports that both are matching and unification compatible.

Example 24. Returning to Example 22, the LFHOL fingerprint sample of position 1 for the term s is **B**, whereas it stays a for t. The $\mathit{gff}_{\mathrm{HO}}$ function now reports that both are matching and unification compatible.

Our main goal with the generalization of fingerprint indexing to LFHOL was to make sure we keep the same level of precision that fingerprints give for FOL. This is challenging because another constraint is that no changes should be made to Tables 5.1 and 5.2, that is to the way fingerprint compatibility is checked. Thus, we could use only the fingerprint values that are available for FOL terms and make sure that they have the right semantics for LFHOL matching and unification without being too coarse-grained.

We managed to keep the compatibility matrices intact by repurposing some fingerprint symbols, without changing their semantics. Note that this extension is graceful since the only notion altered in LFHOL extension, $\mathit{gff}_{\mathrm{HO}}$ function, will give the same results on FOL terms that *gff* function gave. Namely, in FOL case tuple $\overline{x}$ will always have the length of 0 and since variables are not applied in FOL, the condition *head is a variable* degenerates to *term is a variable*. This behavior coincides with FOL *gff*.

5.2.4 hoE Implementation of LFHOL Fingerprint Indexing

In hoE, no changes are performed to compatibility checks nor fingerprint trie traversal. However, the way fingerprints are obtained has changed and the changes are located in the file `TERMS/cte_idx_fp.c`.

Even though we claim that the LFHOL generalization is graceful, we decided to implement LFHOL fingerprinting in a separate procedure from FOL fingerprinting. We think that the overhead that LFHOL generalization incurs is negligible, and will likely replace the old implementation with the new one to keep the code base more manageable.

5.3 Feature-Vector Indices

Most of the indexing techniques work by organizing terms in some kind of structure that efficiently performs an operation between a query term and set of terms.

Algorithm 9 Backward and forward subsumption using Feature Vector indexing

```
procedure FindSubsuming(FVIndexNode $n$, Num $component$, Clause $C$)
    if $n$ is a leaf then
        if any clause in $n$ subsumes $C$ then
            return $True$
        else
            return $False$
    else
        for all $i$ in Range(0, $C.feature\_vector[component]$)
            where node $i$-labeled node is adjacent do
            $neighbor \leftarrow i$-labeled node adjacent to $n$
            if FindSubsuming$(neighbor, component + 1, C)$ then
                return $True$
        return $False$
procedure IsSubsumed(ClauseSet $I$, Clause $C$)
    return FindSubsuming$(I.root, 1, C)$
procedure FindSubsumed(FVIndexNode $n$, Num $component$, Clause $C$)
    $Res \leftarrow \emptyset$
    if $n$ is a leaf then
        return $\{T \mid C$ subsumes a clause $T$ in $n\}$
    else
        for all $i$ in Range($C.feature\_vector[component]$, MaxAdjacent)
            where $i$-labeled node is adjacent to $n$ do
            $neighbor \leftarrow i$-labeled node adjacent to $n$
            $Res \leftarrow Res \cup$ FindSubsumed$(neighbor, component + 1, C)$
        return $Res$
procedure RetrieveSubsumed(ClauseSet $I$, Clause $C$)
    return FindSubsumed$(I.root, 1, C)$
```

However, for some parts of the proof search, indexing techniques that work on sets of clauses are more useful.

One of the most useful simplification inferences is subsumption. Clause C subsumes clause D if there exists an instance of C such that it is a multisubset of D. Put differently, we want to find a substitution σ, such that $\sigma(C) \subseteq D$.

In the main saturation loop of E subsumption is checked in two directions. *Forward subsumption* checks whether any clause in the proof state subsumes the given clause. *Backward subsumption* determines which clauses a given clause subsumes.

The subsumption problem is NP-complete, but Schulz notes that with careful implementation, the worst case is rarely observed [Sch13a]. But the optimized subsumption check implementation only solves the clause-clause subsumption problem, not the clause-against-the-set subsumption problem. One idea, similar to fingerprint indexing, is to collect features of a clause that give us information about subsumption and organize them in a trie structure to provide fast retrieval of compatible clauses. In that way, we could create an imperfect index that will substantially decrease the number of clauses we are going to test for subsumption.

Definition 38. A clause feature function *cff* assigns an integer to a clause. A clause feature function is *compatible with subsumption* if $\mathit{cff}(C) \leq \mathit{cff}(D)$ whenever clause C subsumes clause D.

Definition 39. With $|C|$ we denote the number of literals in a clause C. With $|C|_f$ we denote the number of occurrences of a function symbol f in C. The depth of the deepest occurrence of function symbol f is denoted with d_f and defined recursively:

$$d_f(t) = \begin{cases} 0 & f \text{ does not appear in } t \\ \max\{1, d_f(t_1)+1, \dots, d_f(t_n)+1\} & \text{if } t \equiv f(t_1, \dots, t_n) \\ \max\{d_f(t_1)+1, \dots, d_f(t_m)+1\} & \text{if } t \equiv g(t_1, \dots, t_n) \text{ and } f \neq g \end{cases}$$

d_f is lifted to literals and clauses by taking the maximum of depths in terms that appear in literal and clauses.

Theorem 8. The clause feature functions $|C^+|, |C^-|, |C^+|_f, |C^-|_f, d_f(C^+), d_f(C^-)$, for all function symbols f are compatible with subsumption.

Definition 40. A feature vector function *fvf* for the fixed tuple $\overline{cff} = (cff_1, \dots, cff_n)$ of clause feature functions is defined as $fvf(C) = (cff_1(C), \dots, cff_n(C))$. The result of a feature vector function is called the *feature vector*.

The feature vector for a clause is a vector of integers. By using those vectors as keys in a trie, both forward and backward subsumption compatible clauses can be retrieved efficiently. Namely, if we want to find clauses that might subsume a given clause, we are interested in clauses whose feature vectors are componentwise less than or equal. Similarly, when we want to find clauses that are subsumed by a given clause we are interested in clauses that have feature vectors componentwise greater than or equal to a given clause's feature vector.

In the case of forward subsumption, we are only interested in the existence of a single clause that subsumes a given clause. On the other hand, we want to find all clauses that are subsumed by a given clause. Algorithm 9 determines if a clause is subsumed and if the clause subsumes another clause.

Feature vector tries are implemented in the file `CLAUSES/ccl_fcvindexing.c` in E. The trie implementation is similar to the one for fingerprint indices. E supports different sets of features that show different performance results. Schulz also notes that the order in which we choose features improves performance [Sch13a], which E takes into consideration as well.

Unlike the other two data structures, there were barely any changes that we had to perform to extend the feature vector indices to LFHOL. The only change came from the fact that the depth of LFHOL terms that had variable as head symbol would be calculated wrongly because of the specific encoding we had in place for those terms (see Chapter 3). Furthermore, the artificial `@_var` symbol had to be ignored when counting the number of times each function symbol appears in a clause.

The reason why feature vector index generalizes so gracefully is that it does not compare clauses on the level of the terms they contain. Instead it does a very superficial, yet very efficient, collection of clause aggregates of the function symbol occurrences, depths of symbols and similar. The key observation is that neither for FOL nor for LFHOL applying substitution to a clause can make those aggregates lower. Thus, on the level of the data structure no changes are needed.

Chapter 6

Generalizing the Calculus' Implementation

The performance of a theorem prover both in the competitions and on real-world problems depends on many factors. The choice of the calculus, programming language used, details of the implementation and the data structures interact in subtle ways. In this chapter we are concerned with E's inference rules, how they are implemented and what we had to change to make inferences work in LFHOL setting.

The four core rules of the superposition calculus that are needed to achieve soundness and completeness are called *generating inference rules*. Even though those rules are constrained by the side conditions, they are prolific and generate a large number of clauses. Thus, modern theorem provers spend substantial amounts of time to remove clauses that cannot contribute to the proof – *redundant* clauses.

A clause C is redundant with respect to a clause set S if it logically follows from the ground instances of clauses in S that are smaller than C [Sch02]. A redundant clause can be deleted without affecting the completeness of the core calculus. The inferences that remove redundant clauses are called simplification rules. Those inferences will be shown with a double horizontal line. Unlike generating inference rules that add conclusions to the proof state, simplification rules replace the premises with conclusions in the proof state. If there are no conclusions, premises are simply deleted from the proof state.

Furthermore, some inferences can infer a clause C_m that makes C redundant. This is a two step process of C's removal and without loss of generality we will present only the end results.

Table 6.1 summarizes the changes that we performed and what are the main files in which the changes occurred. It can be used as a roadmap of the following sections.

6.1 Preprocessing

E reads the input in the TPTP format [Sut17a] which allows describing typed first-order formulas (TFF syntax). hoE extends E's input/output module with the support for the HOL THF syntax. As we mentioned in Chapter 2, before we can apply the calculus rules, we have to Skolemize the input problem and convert it to CNF, since the input format allows unrestricted formulas, not just the clauses.

Besides this necessary step, E performs many other optional preprocessing steps that are performed to simplify the problems, while preserving its validity. For example, it removes clauses that can easily be shown tautological.

Most of the preprocessing steps generalize gracefully for LFHOL and there are no changes to be made in terms of the code. However, there are a few exceptions. In this section, we describe the changes we had to make to generalize preprocessing.

Inference	**E source file**	**LFHOL generalization changes**
ER	`CLAUSES/ccl_eqnresolution.c`	Complete LFHOL unification
EF	`CLAUSES/ccl_factor.c`	Complete LFHOL unification
SN/SR	`CLAUSES/ccl_paramod.c`	LFHOL fingerprint indices Prefix LFHOL unification
DR	`CONTROL/cco_eqnresolving.c`	Complete LFHOL unification
RP/RN	`CLAUSES/ccl_rewrite.c`	LFHOL PDTs LFHOL fingerprint indices Prefix LFHOL matching Prefix subterm rewriting
PS/NS/ES	`CLAUSES/ccl_subsumption.c`	LFHOL PDTs Prefix LFHOL matching
Subsumption	`CLAUSES/ccl_subsumption.c`	Complete LFHOL matching LFHOL feature vector indexing LFHOL subsumption order

TABLE 6.1: Summary of inference rules and changes needed for LFHOL

6.1.1 Definition Unfolding

To simplify the proof state before going into the saturation loop, E removes some unit clauses by eagerly applying them wherever possible. In particular, E searches for the unit clauses of the form $f(x_1, \ldots, x_n) = t$ where f does not appear in t and variables that appear in t are from the set $\{x_1, \ldots, x_n\}$, called *definitional clauses*. After finding a definitional clause, E *unfolds* it by matching the left-hand side of the definitional clause onto any subterm that has f as the head symbol, and replaces that subterm with accordingly instantiated term t. It can be proven that eagerly unfolding the definitional clauses does not change the validity of the formula. Thus, it can be used as a nice way to remove certain clauses, and subsequently function symbols, from the proof state.

Unfortunately, definition unfolding makes hoE uncomplete for LFHOL. Intuitively, in FOL definitional clauses can be unfolded and subsequently removed from the proof state because unfolding them is equivalent to performing all of the inferences we could perform with the definitional clause as one of the inference partners. However, this is not true for LFHOL.

When there are applied variables in some of the clauses, applied variables can be unified with the left-hand side of the definitional clause. This means that those clauses can be inference partners with the definitional clause. In addition, unfolding the definitional clause would not unfold the clause onto the applied variable. When the definitional clause is removed from the proof state, unifying the applied variable with the left hand side of the definitional clause will never be tried, yielding the further proof search incomplete.

To resolve this issue, in this version of hoE we disabled the definition unfolding. However, in future we plan to investigate higher-order definition unfolding in greater detail and generalize it gracefully.

6.1.2 Skolemization

Integral part of converting the formula to CNF is Skolemization. For FOL, Skolemization preserves the satisfiability of the formula. For HOL, Skolemization is a more subtle issue.

Namely, one of the most important axioms in HOL is the *Axiom of Choice*. Intuitively, it states that there is a function that chooses an element from every non-empty set. It can be stated as

$$\exists \xi_{(t \to o) \to t} \,.\, \forall P_{t \to o} \,.\, (\exists X_t \,.\, P\ X) \longrightarrow P\ (\xi\ P)$$

The ξ function is called the *choice operator*.

It is thought that the Axiom of Choice is valid in most of the standard HOL models [RV01, Chapter 15]. It is not built into LFHOL and it usually needs special treatment in other higher-order logics.

Skolemization interferes with the Axiom of Choice in subtle ways in HOL. Namely, if we perform the Skolemization from the Chapter 2, the Skolem function symbols we add to the signature might play the role of the choice function. Thus, Skolemization can make formulas that are unprovable without the Axiom of Choice provable. In other words, Skolemization is unsound for LFHOL without the Axiom of Choice.

Example 25. Consider the following formula (stemming from a TPTP problem):

$$(\forall Y_t \,.\, \exists X_t \,.\, f\ X \approx Y) \longrightarrow (\exists G_{t \to t} \,.\, \forall X_t \,.\, f\ (G\ X) \approx X)$$

After converting the negation of the formula to CNF we would get the following set of clauses (that should be equisatisfiable with the original formula):

$$\{\{f\ (s_1\ X_1) \approx X_1\}, \{f\ (X_2\ (s_2\ X_2)) \not\approx s_2\ X_2\}\}$$

where s_1 and s_2 are fresh Skolem function symbols.

After unifying X_2 with s_1 and X_1 with $s_2\ s_1$ and performing the superposition (followed by equality resolution) we would get an empty clause, proving the original formula. The problem that arose here is that variable X_2 was bound to the Skolem function symbol s_1 that was partially applied. Thus, by using Skolemization we forced the existence of function $I_D(s_1)$ (interpretation of s_1), which did not exist in the original problem.

A solution to this problem is to make each Skolem function symbol applied to a number of *necessary arguments* [RV01].In FOL, the Axiom of Choice cannot even be stated, so unrestricted Skolemization is sound for FOL. To restrict the Skolemization, we need to change E in a lot of places including the Skolemization procedure, matching, unification and others.

However, our ultimate goal is to support HOL with the Axiom of Choice which makes making this complicated change unreasonable, since it will be undone in the near future. Thus, we perform unrestricted Skolemization knowing that is unsound for LFHOL without the Axiom of Choice.

6.2 The Generating Inference Rules

Equality resolution (*ER*)

$$\frac{s \not\approx t \lor R}{\sigma(R)}$$

where σ is the mgu of s and t and $\sigma(s \not\approx t)$ is eligible for resolution.

This inference is relatively simple since it is performed on only one premise. Thus, E goes through each maximal negative literal in the clause and tries to unify the left-hand with the right-hand side. If it finds a literal for which this unification succeeds, it creates a new clause without the literal, and with the unifier applied to the rest of the clause.

In hoE, we changed this implementation by running LFHOL unification whenever E is supplied with a higher-order problem. However, in this inference both s and t have to unify the entire terms, not subterms of either term. Thus, the result of the unification cannot have trailing arguments on either side. This is a recurring theme throughout all generating inferences.

Thus, to avoid calling repeatedly first-order unification (`SubstComputeMgu` E function) on FOL problems and higher-order unification (`SubstComputeMguHO` hoE function) on HOL problems and making sure that there are no trailing arguments in all of the places with this constraint, we hid this operation using the function `SubstMguComplete`. This function will report success only when the terms are unified with no trailing arguments. Furthermore, depending on the the type of the problem it will call first- or higher-order unification.

With this interface in place, the differences that we had to make to E were simple. We only had to change the call to `SubstComputeMgu` to `SubstMguComplete`.

Equality factoring (*EF*)

$$\frac{s \approx t \lor u \approx v \lor R}{\sigma(t \not\approx v \lor u \approx v \lor R)}$$

where σ is the mgu of s and t, $\sigma(s) \not< \sigma(t)$ and $\sigma(s \approx t)$ is eligible for paramodulation.

This single-premise inference is more complicated than *ER*, because it involves two literals. E handles this by fixing a maximal literal and tries unifying against all other positive literals. After trying all other positive literals, the next maximal literal is fixed and the process is repeated until no unconsidered maximal literals remain. Like for *ER*, the only difference we made is making sure that only the entire terms are unified. We hid this operation behind the `SubstMguComplete` function, which means we had to change only the call to the unification in *EF*.

Superposition into negative literals (*SN*)

$$\frac{s \approx t \lor S \quad u \not\approx v \lor R}{\sigma(u[p \leftarrow t] \not\approx v \lor S \lor R)}$$

where σ is the mgu of s and $u|_p$, $\sigma(s) \not< \sigma(t), \sigma(u) \not< \sigma(v), \sigma(s \approx t)$ is eligible for paramodulation, $\sigma(u \not\approx v)$ is eligible for resolution and $u|_p \notin V_a$ where a is the type of $u|_p$. s is called the *from-term*, whereas $u|_p$ is called the *to-term*. The literal where s appears is called the *from-literal* and the clause where the from-literal appears is called the *from-clause*. The literal where $u|_p$ appears is called the *to-literal* and the clause where the to-literal appears is called the *to-clause*.

Superposition into positive literals (*SP*)

$$\frac{s \approx t \lor S \quad u \approx v \lor R}{\sigma(u[p \leftarrow t] \approx v \lor S \lor R)}$$

where σ is the mgu of s and $u|_p$, $\sigma(s) \not< \sigma(t), \sigma(u) \not< \sigma(v), \sigma(s \approx t)$ is eligible for paramodulation, $\sigma(u \approx v)$ is eligible for resolution and $u|_p \notin V_a$ where a is the type of $u|_p$. s is called the *from-term*, whereas $u|_p$ is called the *to-term*. The literal where s appears is called the *from-literal* and the clause where the from-literal appears is called the *from-clause*. The literal where $u|_p$ appears is called the *to-literal* and the clause where the to-literal appears is called the *to-clause*.

Superposition inferences are the core inferences of the calculus. From the form of the rules one can see that they are substantially more complex than the rules *EF* and *ER*, since they have two premises.

When E chooses a given clause, it puts it in both the role of the from-clause and the to-clause. In both roles the clause should be compared to all the processed clauses in the proof state. Comparing a clause against all the processed clauses can be very time-consuming, especially when we know that only a handful of clauses will be eligible partners for the superposition inference.

To speed up superposition inferences, E uses the fingerprint indexing. Namely, if a given clause is used as a from-clause, E queries the fingerprint index for subterms in the proof state that are unifiable with all the from terms in the given clause. Furthermore, if a given clause is used as a to-clause, E finds all terms in the proof state that are unifiable with any possible to-subterm.

Superposition inferences posed more challenges in terms of extending the inference rules to LFHOL. A from-term has to be fully unified , whereas the to-term is possibly a subterm of one side of the literal it appears in. Thus, when we call the LFHOL unification procedure, we allow unification of prefixes only in the to-term.

To support this behavior we created the `SubstMguPossiblyPartial` function that does not only return a boolean, but an object of the type `UnificationResult` that has the information about the number of trailing arguments and the term that has trailing arguments. We made sure that upon a successful unification arguments are not remaining on the wrong side – that is in the from-term. Similarly to `SubstMguComplete`, `SubstMguPossiblyPartial` invokes the FOL version of the algorithm if E is run on the FOL problem and makes `UnificationResult` object out of the result of FOL unification.

Note that with the usage of prefix unification we managed to completely avoid creating prefixes of terms and many (possibly expensive) queries to the fingerprint index. The changes that were necessary were running partial unification and making sure the resulting to-term has been constructed properly. By proper construction of the resulting to-term we assume rewriting only the prefix of the to-term and leaving the trailing arguments intact.

Including partial unification was straightforward and included only changing the call to FOL unification to `SubstMguPossiblyPartial` and storing its result. If the unification was successful, the result will contain the information about the number of trailing arguments. This information had to be propagated to the part of the inference engine that creates the resulting to-term. Lastly, the construction of the resulting to-term used this information to rewrite only the unified prefix.

6.3 The Simplification Rules

Rewriting of positive literals (*RP*)

$$\frac{s \approx t \qquad u \approx v \vee R}{\overline{s \approx t \qquad u[p \leftarrow \sigma(t)] \approx v \vee R}}$$

where $\sigma(s) = u|_p, \sigma(s) > \sigma(t)$ and either $u \not\succ v$ or $u \approx v$ is not eligible for resolution or position p is not empty string (ε) or substitution σ is not a variable renaming. $s \approx t$ is called a *rewrite rule*.

Rewriting of negative literals (*RN*)

$$\frac{s \approx t \qquad u \not\approx v \vee R}{\overline{s \approx t \qquad u[p \leftarrow \sigma(t)] \not\approx v \vee R}}$$

where $\sigma(s) = u|_p$ and $\sigma(s) > \sigma(t)$. $s \approx t$ is called a *rewrite rule*.

Rewriting is one of the most important simplification inferences in E. In E, rewriting is performed in two stages. The first stage is performed when a given clause is picked and we want to find all rewrite rules that can rewrite terms in the given clause. This stage is called *forward rewriting*. The second stage is performed when a given clause (possibly after simplifications) is determined to be a unit clause and we want to find out which clauses in the proof state this clause can rewrite – *backward rewriting*.

E supports two levels of forward rewriting, as well as disabling of forward rewriting altogether. The first level uses only unit equations that can be oriented, whereas the second level also uses the non-oriented unit equations, for which orientation of the instantiation is checked upon successful match of either equation side. The intuition behind splitting those two levels is that checking the orientation can be time-consuming, and since reduction orders are closed under substitution, if the equation can be oriented (using a reduction order), every of its instances can be oriented. Thus, for oriented equations the check $\sigma(s) > \sigma(t)$ does not have to be performed. Furthermore, for oriented unit equations only the equations whose larger side is a generalization of a subterm of the given clause need to be retrieved. For non-oriented equations we do not know which side of the equation is a generalization that will give the right orientation beforehand.

Processed oriented unit clauses with a positive literal are represented by the `processed_pos_rules` clause set in the proof state. Those clauses are indexed by the larger side in the equation using PDT. On the other hand, processed non-oriented unit clauses with a positive literal are represented by the `processed_pos_eqns` clause set, which is indexed using both sides of the equation, using a PDT as well.

The process of finding equations that rewrite subterms of a given clause is carried out in a *leftmost-innermost* fashion. That is, for a term $t \equiv f(t_1, \ldots, t_n)$, all subterms of t_1 that are rewritable will be rewritten, then all subterms of t_2 and so on until t_n. Only then will the entire term t be rewritten. Based on the rewrite level, only oriented or both oriented and non-oriented unit equations will be used as potential rewrite rules. This is one of the places where E traverses subterms the way we described it when introducing the LFHOL extension of PDTs (Section 5.1).

On the other hand, when the given clause is (simplified to) a unit clause we want to find terms that can be rewritten by the given clause. E maintains a fingerprint index `bw_rw_index` that stores all rewritable subterms. This index is queried for all the terms that are possible instances of the bigger side of the given clause if it is oriented or either of the sides otherwise. Then the checks for all of the constraints of the rules *RP* or *RN* will be performed and if all checks pass, the clause in which the instance term appears is rewritten and moved from the processed to the unprocessed clause set.

The LFHOL extension of the rewriting support was complex since it included dealing with support for prefix matching that indexing structures are now aware of. This means that the answers to the queries data structures provide are used differently.

Namely, when querying PDTs for generalizations, it might be the case that some arguments are trailing in the query term. We had to take care of that and make sure that we construct the rewritten term correctly.

In hoE, the function that returns matches from PDT had to be altered. It now returns the `MatchInfo` object that, alongside the information about the clause that can be used as a rewrite rule, also gives the information about the number of trailing arguments. Moreover, construction of the newly rewritten term is depending on the number of trailing arguments and it is performed in the `MakeRewrittenTerm` function.

Furthermore, next to the already described changes to the fingerprint indexing, since it is a non-perfect indexing technique, we had to run LHFOL matching algorithm after retrieving candidates in the backward-rewriting stage. In this case, the query term had to be matched fully, but the candidates could have trailing arguments. To allow this behavior we used `SubstMatchPossiblyPartial`, which allows trailing arguments in the target. Again, we combined it with `MakeRewrittenTerm` to construct the rewritten term respecting the trailing arguments. This is one of the places where our subterm optimization came in handy – the code changes outside PDTs were needed only in a few places. Furthermore, terms that are prefixes of other terms stored in the PDT are traversed in the shortest-first fashion. This corresponds to the existing leftmost-innermost semantics of E's rewrite subterm traversal loop.

Destructive equality resolution (*DR*)

$$\frac{x \not\approx s \vee R}{\sigma(R)}$$

where σ is the mgu of x and s.

DR inference is one of the simplest inferences in terms of implementation. By default, E performs *DR* only if s is a variable, or if strong *DR* is enabled, it performs it nonetheless. For the LFHOL generalization, we had to only make sure that upon successful unification no remaining term is left in the right hand side of the literal that is involved in the *DR*.

Positive simplify-reflect (*PS*)

$$\frac{s \approx t \qquad u[p \leftarrow \sigma(s)] \not\approx u[p \leftarrow \sigma(t)] \vee R}{s \approx t \qquad R}$$

Negative simplify-reflect (*NS*)

$$\frac{s \not\approx t \qquad \sigma(s \approx t) \vee R}{s \approx t \qquad R}$$

Equality subsumption (*ES*)

$$\frac{s \approx t \qquad u[p \leftarrow \sigma(s)] \approx u[p \leftarrow \sigma(t)] \vee R}{s \approx t}$$

Equality subsumption is also called *unit subsumption*.

In all three inferences we are interested in finding a negative or positive unit equation (called *simplifier*) that can be matched onto a literal in the given clause. However, in case of *PS* and *ES*, the simplifier does not have to be matched onto the entire literal, but possibly to the pair of terms appearing in the same term context in one literal (called *candidate pair*).

When performing *PS* and *ES* E queries the set of positive unit equations in the proof state, `processed_pos_eqns`, for possible simplifiers and chooses terms that appear in the same context in one literal as candidate pairs. For *NS* the set of negative unit equations, `processed_neg_eqns`, will be queried and only terms appearing at the top level in the literal will be considered as the candidate pair. In other words, for *NS* the term context has to be empty. Both of the sets are indexed by both sides of the equation using PDTs.

Example 26. Suppose that the literal $f(a,c) \approx f(b,c)$ is picked as the candidate pair in the given clause, and we want to test for equality subsumption. If no simplifier was found for the candidate pair $(f(a,c), f(b,c))$, we must try subterms that appear in the same context as the new candidate pair. The terms a and b appear in the same context, so the next equality subsumption check will be performed for (a,b). If the original equation was $f(a,c) \approx f(b,d)$, the terms a and b would not appear in the same context which means no subterm will be chosen as new candidate pair.

The LFHOL extension of *NS* was straightforward. In *NS*, the candidate pair had to be matched onto completely (without trailing arguments). For that reason, we had to make sure that no arguments are remaining when the PDT is queried for the generalization of the first term in the candidate pair. Furthermore, the remaining term of the simplifier has to be matched onto the second part of the given candidate pair. This is done using `SubstMatchComplete`, that performs the check that no arguments remain.

The LFHOL extension of *PS* and *ES* was more involved. This stems from the complex subterm traversal scheme that complicates even further when prefix subterms have to be considered as well. However, the support for prefix matching in PDTs and in the LFHOL matching procedure proved helpful both in terms of the algorithm complexity and the amount of code we had to change.

The process of finding simplifiers for rules *PS* and *ES* is described in Algorithm 10. It iterates through subterms in the leftmost-outermost fashion.

In LFHOL, each term has prefix subterms which have to be traversed as well. First-order Algorithm 10 fails to capture this prefix matching. Furthermore, when the simplifier for the prefix of the candidate pair is found, the remaining arguments in both of the elements in the candidate pair have to be equal for prefix to appear in the same term context, as Example 27 illustrates.

Example 27. Suppose $f\,a\,b\,c \approx g\,(h\,a)\,c$ is the candidate pair and the found simplifier is $f\,X_t\,Y_t \approx g\,(h\,X_t)$. Then, we can conclude that the candidate pair is subsumed by the simplifier since the simplifier matches a prefix of the candidate pair and the trailing arguments of the candidate pair are the same. If the candidate pair was $f\,a\,b\,c \approx g\,(h\,a)\,d$ we could not make the same conclusion, since the context in which prefix subterms occur would not be the same.

One could now wonder what changes are needed to make Algorithm 10 traverse prefix subterms and to make sure they appear in the same context. The solution that would be the easiest from the implementation point of view would be to create prefix terms explicitly and query the unit clause set for simplifiers. Obvious drawbacks are

Algorithm 10 Finding simplifiers

```
procedure FindSimplifier(ClauseSet $set$, Term $s$, Term $t$)
    for all $eq$ in $set$ whose one side generalizes $s$ with substitution $\sigma$ do
        $o\_side \leftarrow$ other side of $eq$
        if $\sigma(o\_side)$ can be matched onto $t$ then
            return $True$
    if $s \equiv f(s_1, \ldots, s_n)$ and $t \equiv f(t_1, \ldots, t_n)$ then
        $cand\_pair \leftarrow$ nil
        for $i = 1$ to $n$ do
            if $s_i \not\equiv t_i$ then
                if $cand\_pair =$ nil then
                    $cand\_pair \leftarrow (s_i, t_i)$
                else
                    return $False$
        if $cand\_pair \equiv (s', t')$ then
            return FindSimplifier$(set, s', t')$
        else
            return $False$
    return $False$
```

that we can have to create $O(n)$ prefixes where n is the number of term arguments and query the clause set $O(n)$ times.

First, if s and t have the same type and the same head function symbol, they have to be applied to the same number of arguments. This means that subterm traversal limits at line 8 of Algorithm 10 do not have to be changed. Second, the LFHOL extension of PDTs allows prefix matching, so we can easily obtain the length of the only prefix that can be matched by the given simplifier. Third, it is clear that, since both elements of the candidate pair have the same type, the number of the remaining arguments in both elements of the pair when simplifier is matched onto them must be the same. Lastly, after the successful match, we need to make sure that the remaining arguments in both parts of the given pair are the same to make sure that inferences operate on subterms that occur in the same term context.

In Algorithm 11 described changes are put in place to enable support for LFHOL version of *PS* and *ES*. Thanks to prefix LFHOL PDT matching, we avoided creating prefix subterms altogether.

In conclusion, this made the implementation of the *PS* and *ES* entirely graceful. In particular, due to avoided prefix term creation, the algorithm time complexity remained the same. Additionally, the simplifier retrieval algorithm had to be changed only in a few places.

In addition, E has the *strong* version of *ES* and *PS* in which the search for matching equation is done for each different argument pair of the literal. However there is almost no difference in LFHOL generalization of this inference – if any of the trailing arguments are pairwise different, we just have to perform lookup form matching equation recursively.

For completeness we give the definition of the strong version of *ES* and *PS* rules:

Algorithm 11 Finding simplifiers in LFHOL

```
1: procedure FINDSIMPLIFIERHO(ClauseSet set, Term s, Term t)
2:     for all eq in set whose one side generalizes s
3:         with substitution $\sigma$ and $r$ trailing arguments do
4:         o_side ← other side of eq
5:         if $\sigma$(o_side) can be matched onto $t[: -r]$ then
6:             if remaining $r$ arguments in $s$ and $t$ are the same then
7:                 return True
8:             else
9:                 return False
10:    if $s \equiv f(s_1, \ldots, s_n)$ and $t \equiv f(t_1, \ldots, t_n)$ then
11:        cand_pair ← nil
12:        for $i = 1$ to $n$ do
13:            if $s_i \not\equiv t_i$ then
14:                if cand_pair = nil then
15:                    cand_pair ← $(s_i, t_i)$
16:                else
17:                    return False
18:        if cand_pair ≡ $(s', t')$ then
19:            return FINDSIMPLIFIERHO$(set, s', t')$
20:        else
21:            return False
22:    return False
```

Strong positive simplify-reflect (*SPS*)

$$\frac{s_1 \approx t_1 \;\ldots\; s_n \approx t_n \qquad u[p_1 \leftarrow \sigma(s_1), \ldots, p_n \leftarrow \sigma(s_n)] \not\approx u[p_1 \leftarrow \sigma(s_1), \ldots, p_n \leftarrow \sigma(s_n)] \vee R}{s_1 \approx t_1 \;\ldots\; s_n \approx t_n \qquad R}$$

Strong equality subsumption (*SES*)

$$\frac{s_1 \approx t_1 \;\ldots\; s_n \approx t_n \qquad u[p_1 \leftarrow \sigma(s_1), \ldots, p_n \leftarrow \sigma(s_n)] \approx u[p_1 \leftarrow \sigma(s_1), \ldots, p_n \leftarrow \sigma(s_n)] \vee R}{s_1 \approx t_1 \;\ldots\; s_n \approx t_n}$$

In both inferences it is crucial that the positions $p_1, \ldots, p_n$ are non-overlapping (no position p_i is a prefix of different position p_j). Strong versions of positive simplify-reflect and equality subsumption are disabled by default and can be enabled using E command-line options.

6.4 Subsumption

Subsumption is one of the most important simplifying inferences because it can remove from 50% up to 95% of all the clauses in the proof state [Sch13a]. We briefly discussed subsumption in Chapter 5, when we discussed feature vector indexing and in this chapter when we discussed unit subsumption.

The main idea behind the subsumption inference is that if in the proof state one has a clause C that is more general than the clause D, clause D can be safely removed. C is more general than D if some instance of C is a multisubset of D, as the following rule shows:

$$\frac{C \qquad \sigma(C) \vee \sigma(R)}{C}$$

A superset of the clauses that can be subsumed by a given clause or can subsume a given clause is returned using the feature vector indexing. We described this process in Section 5.3. In what follows we will delve deeper in the procedure that checks if one clause subsumes another.

Before E tries to check if a clause (called the *subsumer*) subsumes another one (called the *(subsumption) candidate*) it makes sure that both clauses are *subsumption-ordered*. The subsumption order $\rhd$ is an ad hoc order on literals whose purpose is to make sure we only consider literals in subsumer that can potentially be matched onto literals in candidate.

$l_1 \rhd l_2$ if any of the following conditions is satisfied:

1. l_1 is positive and l_2 is negative.
2. Both l_1 and l_2 are of same polarity and l_1 is equational, whereas l_2 is non-equational. Non-equational literals are the literals of the form $P(t_1, \ldots, t_n) \approx \mathbf{T}$ or $P(t_1, \ldots, t_n) \not\approx \mathbf{T}$.
3. l_1 and l_2 have the same polarity, both are non-equational and function code of the left-hand side in l_1 is larger than the function code of the left-hand side in l_2.

It can be shown that literal l_1 can be matched on l_2 only if $l_1 \not\rhd l_2$ and $l_2 \not\rhd l_1$. Thus, as soon as we reach a smaller literal in the candidate list we can return a failure. If no failure is observed, using the Algorithm 12 E checks whether the subsumer can indeed subsume the candidate. Before calling the Algorithm 12 we must sort the literals in both the subsumer and the candidate using $\rhd$, from larger to smaller.

Algorithm 12 has exponential time complexity in the number of literals of both the subsumer and the candidate. The introduction of the subsumption order $\rhd$ on literals helps keeping the number of matching tries low, speeding up the algorithm. Note that as soon as we observe a smaller literal in the candidate list, we can report the failure, possibly pruning large parts of the search space.

Algorithm 12 Subsumption using the subsumption ordering

```
 1: procedure ListSubsume(EqList $subsumer$, EqList $candidate$, Set $matched$)
 2:     if $subsumer$ = nil then
 3:         return $True$
 4:     else
 5:         $subs\_eq \leftarrow subsumer\,.\,val$
 6:         for all $cand\_eq$ in $candidate$ do
 7:             if $cand\_eq$ is not in $matched$ then
 8:                 if $subs\_eq \rhd cand\_eq$ then
 9:                     return $False$
10:                 else if $cand\_eq \not\rhd subs\_eq$ then
11:                     $matched \leftarrow matched \cup \{cand\_eq\}$
12:                     if $cand\_eq\,.\,lhs$ matches $subs\_eq\,.\,lhs$
13:                         with matching substitution $\sigma$ then
14:                         if $\sigma(cand\_eq\,.\,rhs)$ matches $subs\_eq\,.\,rhs$
15:                             with matching substitution $\tau$ then
16:                             if ListSubsume($\tau \circ \sigma(subsumer\,.\,next)$,
17:                                 $\tau \circ \sigma(candidate)$, $matched$) then
18:                                 return $True$
19:                     if $cand\_eq\,.\,rhs$ matches $subs\_eq\,.\,lhs$
20:                         with matching substitution $\sigma$ then
21:                         if $\sigma(cand\_eq\,.\,lhs)$ matches $subs\_eq\,.\,rhs$
22:                             with matching substitution $\tau$ then
23:                             return ListSubsume($\tau \circ \sigma(subsumer\,.\,next)$,
24:                             $\tau \circ \sigma(candidate)$, $matched$)
25:                     $matched \leftarrow matched \setminus \{cand\_eq\}$
26:         return $False$
```

The LFHOL extension of the ordered subsumption must take into account the applied variables that act as wildcards in the matching procedure since a variable can match any term of the same type. Thus, matching a literal that has an applied variable at the top level must be tried on every literal of the same polarity and kind (equational or non-equational). Note that in the FOL case, predicate variables could never appear, so this case was not considered at all. We extend $\rhd$ to $\rhd_{\mathrm{HO}}$ by refining the third condition in the definition of $\rhd$. In particular, $l_1 \rhd_{\mathrm{HO}} l_2$ if any of the following conditions is satisfied:

1. A positive literal is larger than a negative literal.
2. An equational literal is larger than a non-equational literal.
3. l_1 and l_2 have the same polarity, both are non-equational, neither one has an (applied) variable as the left-hand side and the function code of the left-hand side in l_1 is larger than the function code of the left-hand side in l_2.

The only difference between $\rhd$ and its LFHOL counterpart $\rhd_{\mathrm{HO}}$ is the treatment of variables that appear at the top level in non-equational literals. By using $\rhd_{\mathrm{HO}}$ in Algorithm 12, matching applied variables is tried against every non-equational literal of the same polarity. Furthermore, for the non-equational literals that have no top-level variables only the literals that have the same top-level function code are tried as possible matching targets. This fully coincides with the FOL solution, which makes $\rhd_{\text{HO}}$ entirely graceful. In other words, if we used $\rhd_{\text{HO}}$ on FOL problems, it would behave exactly like $\rhd$.

Delaying the matching of applied variables would be a good way to speed up subsumption procedure. Namely, since applied variables have already described wildcard behavior it is more likely that a literal that has applied variables at the top level will match a target. Thus, to prune out the search space we want to try literals that are not as robust with respect to matching first – that is, we would like to try the literals with no applied variables as early as possible. To that end, while performing the subsumption sort, in both the subsumer and the candidate we always put applied variables at the end of their partition.

This optimization is both graceful and efficient for LFHOL. Since top-level variables cannot appear in non-equational literals in FOL case, new sorting procedure does not change FOL subsumption-sort behavior. On the other hand, it is a heuristic that might improve LFHOL subsumption performance.

Chapter 7

Conclusion

Extending a first-order prover to a fragment of HOL means making a set of very delicate changes to the prover from both theoretical and engineering perspective. A priori, it is not clear if a prover can be gracefully generalized. In this chapter, we give an overview of the changes we made and argue that our generalization is entirely graceful.

7.1 Results

The main question we posed in Chapter 1 is

> *Is there a way to extend a state-of-the-art first-order theorem prover to* **HOL** *in a way that its* **performance on first-order problems remains the same?**

To answer this question we investigated E from both an engineering perspective (e.g. representation of terms and types) and a more abstract, theoretical perspective (e.g. generalization of indexing data structures). In this section, we lay out our main findings.

Terms and Types We have shown that with some effort, the central data structure used in an ATP, the term structure, can be generalized so that all the invariants that previously held for FOL problems are preserved when hoE is supplied with a FOL problem. However, we observed that there are some corner cases where the invariants E has in place have to be generalized to make E work seamlessly with LFHOL. In particular, applied variables challenged us to change the term representation so that the head symbol of an applied variable has a specially designated function code.

Generalization of types was performed by rewriting the type module from the ground up. We have shown that it is possible to implement an HOL type system in a first-order prover in a way that keeps the same time complexity. Furthermore, our type system uses flattened representation that enables efficient treatment of types in LFHOL as well.

Matching and Unification Faced with the problem of large number of prefixes LFHOL terms exhibit, we have spent a substantial amounts of time on trying to find efficient matching and unification algorithms for LFHOL. The algorithms we came up with are able to determine the only prefix that can be matched onto or unified with, with no need for the caller to create exact prefixes. Furthermore, the algorithms keep the same time complexity as in E for both LFHOL and FOL terms. This makes our implementation of LFHOL matching and unification not only graceful but also efficient for LFHOL terms.

Indexing Data Structures E employs advanced data structures to speed up search for the proof. Generalizing the data structures was the most time-consuming part of this project for many reasons. First, all of those data structures have implicit or explicit assumption that the terms they operate on are FOL terms. Second, indexing data structures can be hard to implement and the modules implementing them can be lengthy and contain complicated and hard-to-understand code. Last, they are used in many parts of the proof search, which makes locating bugs hard.

Perfect discrimination trees were the most complicated data structure to generalize. We managed to keep the time complexity for retrieving FOL terms the same as in the original E. Moreover, the order in which terms are retrieved and the way discrimination tree is traversed remained the same. For LFHOL terms, we implemented our prefix-matching optimizations and kept the same time complexity as for the FOL terms.

Extension of fingerprint indexing was not hard from the engineering perspective, but coming up with a graceful extension and reinterpretation of fingerprints was where the majority of the time spent on fingerprint indexing went. Our LFHOL fingerprinting is precise for LFHOL terms and keeps exactly the same granularity of precision for FOL terms.

Extension of feature vector indexing was the easiest generalization we had to perform. From the theoretical perspective, all the subsumption-compatible aggregate features of FOL clauses carry over to LFHOL clauses. On the other hand, from the engineering perspective we had to ignore the special applied variable function code from function code occurrence and depth computations. This change was not time consuming, but required deep understanding of how E feature vector indexing works.

Calculus Generalization The biggest concern we had with the superposition calculus and various simplification rules implemented in E was that many of them work with subterms of a literal. Since the number of subterms in a LFHOL term is double the number of the subterms of the corresponding FOL term, LFHOL prefix traversal could become a bottleneck.

Thanks to our prefix-matching and prefix-unification optimizations that are in place for both the matching and unification procedures and indexing data structures, we managed to avoid traversing and creating subterms other than the ones for FOL terms. However, from the engineering perspective there is still work to be done since every dereferencing of applied variables expands to a new term, which is wasteful.

7.2 Related Work

To the best of our knowledge, hoE is the first ATP that was designed to support exactly LFHOL. However, there is a number of ATPs that support full HOL. Leo-III [Eit+17] and Satallax [GMS12] are perhaps the most popular fully automatic solutions for HOL ATP.

They are built from the ground up for HOL which means that, for example, Leo-III's calculus encodes unification as a clause in an inference which comes as a cost for FOL reasoning. Thus, Leo-III is substantially slower than E on FOL problems. It tries to call external first-order reasoners in parallel to speed up proof search whenever it can, but it is still in infancy compared to the state-of-the-art FOL reasoners. On the other hand, hoE has first-order reasoning built-in natively and performs some higher-order reasoning when needed. Thus, it has finer granularity of control of interleaving first-order and higher-order reasoning.

Furthermore, Satallax does not even support stating problems in FOL syntax (TPTP TFF, FOF or CNF syntax). hoE on the other hand is based on E, first-order prover and behaves like E on FOL problems, so it allows for a more smooth transition to LFHOL. However, hoE falls behind Leo-III and Satallax in terms of its deduction abilities on HOL problems. hoE is not able to generate lambdas at the moment, meaning that is able to prove a strict subset of formulas Leo-III and Satallax can.

From the more theoretical perspective, Blanchette et al. [Bec+17]; [BWW17] have tackled the issue of term orders that are suited for LFHOL terms. Namely, they extended both KBO and LPO to LFHOL gracefully. We are unaware of other theoretical treatments of LFHOL issues.

7.3 Future Work

The main line in which we are going to continue our work is creating a stable version of hoE for which we can give reasonable assurance that it contains no critical bugs.

To be able to answer our research question with more evidence than just theoretically same worst-case time complexity boundaries for FOL and LFHOL algorithms we have to perform experiments on large sets of problems. We haven't performed experiments on large problem corpus.

Experiment results will guide us in optimizing the right parts of hoE. Namely, based on the results from the experimentation phase we can get insight in what are the bottlenecks in LFHOL reasoning. Then, we can revise some of the current design decisions and improve hoE further.

So far, we have mostly been interested in generalizing FOL indexing techniques. Libal and Steen have extended substitution trees, an indexing technique not covered in this thesis, to full HOL terms [SWB16]. It might be interesting to see how this data structure can be specialized for LFHOL terms and whether inclusion of substitution trees would hinder hoE's performance on FOL problems.

Lastly, we want to include support for full HOL in future versions of hoE. Currently, no plan or timeline has been established as to when more HOL features will be added to hoE. However, support for full HOL and keeping E's performance on FOL is our ultimate goal.

Bibliography

[And86] Peter B. Andrews. *An introduction to mathematical logic and type theory - to truth through proof*. Computer science and applied mathematics. Academic Press, 1986. ISBN: 978-0-12-058535-9.

[Bec+17] Heiko Becker, Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand. "A Transfinite Knuth-Bendix Order for Lambda-Free Higher-Order Terms". In: *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*. 2017, pp. 432–453.

[Bez+03] M. Bezem, J.W. Klop, R. de Vrijer, and Terese. *Term Rewriting Systems*. Cambridge Tracts in Theoretica. Cambridge University Press, 2003. ISBN: 9780521391153. URL: https://books.google.nl/books?id=oe3QKzhFEBAC.

[BG90] Leo Bachmair and Harald Ganzinger. "On Restrictions of Ordered Paramodulation with Simplification". In: *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*. 1990, pp. 427–441. DOI: 10.1007/3-540-52885-7_105. URL: https://doi.org/10.1007/3-540-52885-

[BK98] Christoph Benzmüller and Michael Kohlhase. "Extensional Higher-Order Resolution". In: *Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*. 1998, pp. 56–71.

[BWW17] Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand. "A Lambda-Free Higher-Order Recursive Path Order". In: *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 2017, pp. 461–479.

[CL73] Chin-Liang Chang and Richard C. T. Lee. *Symbolic logic and mechanical theorem proving*. Computer science classics. Academic Press, 1973. ISBN: 978-0-12-170350-9.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[Eit+17] Thomas Eiter, David Sands, Geoff Sutcliffe, and Andrei Voronkov, eds. *IWIL@LPAR 2017 Workshop and LPAR-21 Short Presentations, Maun, Botswana, May 7-12, 2017*. Vol. 1. Kalpa Publications in Computing. EasyChair, 2017. URL: https://easychair.org/publications/volume/LPAR-21S.

[GMS12] Bernhard Gramlich, Dale Miller, and Uli Sattler, eds. *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*. Vol. 7364. Lecture Notes in Computer Science. Springer, 2012.

[Ker91] Manfred Kerber. "How to Prove Higher Order Theorems in First Order Logic". In: *Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991*. 1991, pp. 137–142. URL: http://ijcai.org/Proceedings/91-1/Papers/023.pdf.

[Knu98] Donald Ervin Knuth. *The art of computer programming, , Volume III, 2nd Edition*. Addison-Wesley, 1998. ISBN: 0201896850. URL: http://www.worldcat.org/oclc/31

[LS01] Bernd Löchner and Stephan Schulz. "An evaluation of shared rewriting". In: *Proceedings of the Second International Workshop on Implementation of Logics, Technical Report MPI-I-2001-2-006*. 2001, pp. 33–48.

[Löc06] Bernd Löchner. "Things to Know when Implementing KBO". In: *J. Autom. Reasoning* 36.4 (2006), pp. 289–310.

[McC92] William McCune. "Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval". In: *J. Autom. Reasoning* 9.2 (1992), pp. 147–167.

[McC97] William McCune. "Solution of the Robbins Problem". In: *J. Autom. Reasoning* 19.3 (1997), pp. 263–276.

[NR92] Robert Nieuwenhuis and Albert Rubio. "Basic superposition is complete". In: *ESOP '92*. Ed. by Bernd Krieg-Brückner. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 371–389. ISBN: 978-3-540-46803-5.

[RV01] Alan Robinson and Andrei Voronkov, eds. *Handbook of Automated Reasoning*. Elsevier Science Publishers B. V., 2001. ISBN: 0-444-50812-0.

[SBP13] Nik Sultana, Jasmin Christian Blanchette, and Lawrence C. Paulson. "LEO-II and Satallax on the Sledgehammer test bench". In: *J. Applied Logic* 11.1 (2013), pp. 91–102.

[Sch02] Stephan Schulz. "E - a brainiac theorem prover". In: *AI Commun.* 15.2-3 (2002), pp. 111–126. URL: http://content.iospress.com/articles/ai-communications/

[Sch12] Stephan Schulz. "Fingerprint Indexing for Paramodulation and Rewriting". In: *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*. 2012, pp. 477–483.

[Sch13a] Stephan Schulz. "Simple and Efficient Clause Subsumption with Feature Vector Indexing". In: *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*. 2013, pp. 45–67.

[Sch13b] Stephan Schulz. "System Description: E 1.8". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*. 2013, pp. 735–743. DOI: 10.1007/978-3-642-45221-5_49. URL: https://doi.org/10.1007/978-3-642-4

[Sut17a] G. Sutcliffe. "The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0". In: *Journal of Automated Reasoning* 59.4 (2017), pp. 483–502.

[Sut17b] Geoff Sutcliffe. "The CADE-26 automated theorem proving system competition - CASC-26". In: *AI Commun.* 30.6 (2017), pp. 419–432. DOI: 10.3233/AIC-170744. URL: https://doi.org/10.3233/AIC-170744.

[SWB16] G.-M. Greuel, T. Koch, P. Paule, and A. Sommese, eds. *Agent-Based HOL Reasoning*. Vol. 9725. LNCS. Berlin, Germany: Springer, 2016, pp. 75–81. ISBN: 978-3-319-42431-6. DOI: 10.1007/978-3-319-42432-3_10. URL: http://christoph-benzmueller.de/papers/C56.pdf.

Declaration of Authorship

I hereby declare that this master thesis was independently composed and authored by myself.

All content and ideas drawn directly or indirectly from external sources are indicated as such. All sources and materials that have been used are referred to in this thesis.

The thesis has not been submitted to any other examining body and has not been published.

Amsterdam, 29.01.2018.

Signed: Petar Vukmirović