Extending a Brainiac Prover to Lambda-Free Higher-Order Logic (Technical Report)

Petar Vukmirović[1(✉)], Jasmin Christian Blanchette[1,2],
Simon Cruanes[3], and Stephan Schulz[4]

[1] Vrije Universiteit Amsterdam, Amsterdam, the Netherlands
p.vukmirovic@vu.nl
[2] Max-Planck-Institut für Informatik, Saarland Informatics Campus,
Saarbrücken, Germany
[3] Aesthetic Integration, Austin, Texas, USA
[4] DHBW Stuttgart, Stuttgart, Germany

Abstract. Decades of work have gone into developing efficient proof calculi, data structures, algorithms, and heuristics for first-order automatic theorem proving. Higher-order provers lag behind in terms of efficiency. Instead of developing a new higher-order prover from the ground up, we propose to start with the state-of-the-art superposition-based prover E and gradually enrich it with higher-order features. We explain how to extend the prover's data structures, algorithms, and heuristics to λ-free higher-order logic, a formalism that supports partial application and applied variables. Our extension outperforms the traditional encoding and appears promising as a stepping stone towards full higher-order logic.

1 Introduction

Superpostition-based provers, such as E [42], SPASS [52], and Vampire [26], are among the most successful first-order reasoning systems. They serve as backends in various frameworks, including software verifiers (Why3 [22]), automatic higher-order theorem provers (Leo-III [44], Satallax [16]), and one-click "hammers" in proof assistants (HOLyHammer for HOL Light [24], Sledgehammer for Isabelle [34]). Decades of research have gone into refining calculi, devising efficient data structures and algorithms, and developing heuristics to guide proof search [43]. This work has mostly focused on first-order logic with equality, with or without arithmetic [21, 25, 36].

Research on higher-order automatic provers has resulted in systems such as LEO [9], LEO-II [11], and Leo-III [44], based on paramodulation, and Satallax [16], based on analytic tableaux and SAT solving. These provers feature a "cooperative" architecture, pioneered by LEO: They are full-fledged higher-order provers that regularly invoke an external first-order prover with a low time limit as a terminal procedure, in an attempt to finish the proof quickly using only first-order reasoning. However, the first-order backend will succeed only if all the necessary higher-order reasoning has been performed, meaning that much of the first-order reasoning is carried out by the slower higher-order prover. As a result, this architecture leads to suboptimal performance on first-order problems and on problems with a large first-order component, such as those that often arise in interactive verification [46]. For example, at the 2017 installment of the CADE ATP System Competition (CASC) [49], Leo-III, using E as a backend, proved 652 out of 2000 first-order problems in the Sledgehammer division, compared with 1185 for E on its own, and 1433 for Vampire.

To obtain better performance, we propose to start with a competitive first-order prover and extend it to full higher-order logic one feature at a time. Our goal is a *graceful* extension, so that the system behaves as before on first-order problems, performs mostly like a first-order prover on typical, mildly higher-order problems, and scales up to arbitrary higher-order problems, in keeping with the zero-overhead principle: *What you don't use, you don't pay for.*

As a stepping stone towards full higher-order logic, we initially restrict our focus to a higher-order logic without λ-expressions (Sect. 2). Compared with first-order logic, its distinguishing features are partial application and applied variables. This formalism is rich enough to express the recursive equations of higher-order combinators, such as the map operation on finite lists:

$$\mathsf{map}\ f\ \mathsf{nil} \approx \mathsf{nil} \qquad \mathsf{map}\ f\ (\mathsf{cons}\ x\ xs) \approx \mathsf{cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)$$

Our vehicle is E [39,42], a prover developed primarily by Schulz. It is written in C and offers good performance, with the emphasis on "brainiac" heuristics rather than raw speed. E regularly scores among the top systems at CASC, and usually is the strongest open-source[1] prover in the relevant divisions. It also serves as a backend for competitive higher-order provers. We refer to our extended version of E as Ehoh. It corresponds to E version 2.3 configured with the option `-enable-ho`. An earlier version is described in Vukmirović's MSc thesis [51].

The three main challenges are generalizing the type and term representation (Sect. 3), the unification and matching algorithms (Sect. 4), and the indexing data structures (Sect. 5). We also adapted the inference rules (Sect. 6), the heuristics (Sect. 7), and the preprocessor (Sect. 8).

A novel aspect of our work is *prefix optimization.* Higher-order terms contain twice as many proper subterms as first-order terms; for example, the term f (g a) b contains not only the argument subterms g a, a, b but also the "prefix" subterms f, f (g a), g. Many operations, including superposition and rewriting, require traversing all subterms of a term. Using prefix optimization, the prover traverses subterms recursively in a first-order fashion, considering all the prefixes of the current subterm together, at no additional cost. Our experiments (Sect. 9) show that Ehoh is effectively as fast as E on first-order problems and can also prove higher-order problems that do not require synthesizing λ-terms. As a next step, we plan to add support for λ-terms and higher-order unification.

[1] `http://wwwlehre.dhbw-stuttgart.de/~sschulz/WORK/E_DOWNLOAD/V_2.3/`

2 Logic

Our logic corresponds to the intensional λ-free higher-order logic (λfHOL) described by Bentkamp, Blanchette, Cruanes, and Waldmann [8, Sect. 2]. Another possible name for this logic would be "applicative first-order logic." Extensionality can be obtained by adding suitable axioms [8, Sect. 3.1].

A type is either an atomic type ι or a function type $\tau \to \upsilon$, where τ and υ are themselves types. Terms, ranged over by s, t, u, v, are either *variables* $x, y, z, \ldots$, (*function*) *symbols* $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{f}, \mathsf{g}, \ldots$ (often called "constants" in the higher-order literature), or binary applications of the form $s\ t$. Application associates to the left, whereas $\to$ associates to the right. The typing rules are as for the simply typed λ-calculus: If s has type $\tau \to \upsilon$ and t has type τ, then $s\ t$ has type υ. A term's *arity* is the number of extra arguments it can take; thus, if f has type $\iota \to \iota \to \iota$ and a has type ι, then f is binary, $\mathsf{f}\ \mathsf{a}$ is unary, and $\mathsf{f}\ \mathsf{a}\ \mathsf{a}$ is nullary.

Terms have a unique "flattened" decomposition of the form $\zeta\ s_1\ \ldots\ s_m$, where ζ, the *head*, is a variable x or symbol f, and $s_1, \ldots, s_m$, the arguments, are arbitrary terms. We abbreviate tuples $(a_1, \ldots, a_m)$ to $\overline{a_m}$ or $\overline{a}$; abusing notation, we write $\zeta\ \overline{s_m}$ for the curried application $\zeta\ s_1\ \ldots\ s_m$.

An equation $s \approx t$ corresponds to an unordered pair of terms. A literal L is an equation $s \approx t$, where s and t must have the same type, or its negation, written $s \not\approx t$. Predicate symbols are encoded as terms of a distinguished Boolean type; for example, $\mathsf{even}(n)$ is encoded as $\mathsf{even}(n) \approx \mathsf{true}$. Clauses C, D are finite multisets of literals, interpreted disjunctively: $L_1 \vee \cdots \vee L_n$. E and Ehoh clausify the input as a preprocessing step.

Substitutions σ are partial functions of finite domain from variables to terms, written $\{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$, where each x_i has the same type as s_i. The notation $\sigma[x \mapsto s]$ represents the substitution that maps x to s and that otherwise coincides with σ. Applying a substitution σ to a variable beyond σ's domain is the identity. Applying a substitution to a term t applies it homomorphically to t's variables. Composition $(\sigma \circ \sigma')(t)$ is defined as $\sigma(\sigma'(t))$.

A well-known technique to support λfHOL using first-order reasoning systems is to employ the *applicative encoding*. Following this scheme, every n-ary symbol is converted to a nullary symbol, and application is represented by a distinguished binary symbol @. For example, the λfHOL term $\mathsf{f}\ (x\ \mathsf{a})\ \mathsf{b}$ is encoded as the first-order term $@(@(\mathsf{f}, @(x, \mathsf{a})), \mathsf{b})$. However, this representation is not graceful; it clutters data structures and impacts proof search in subtle ways, leading to poorer performance, especially on large benchmarks. In our empirical evaluation, we find that for some prover modes, the applicative encoding incurs a 15% decrease in success rate (Sect. 9). For these and further reasons (Sect. 10), it is not an ideal basis for higher-order reasoning.

3 Types and Terms

The term representation is a fundamental question when building a theorem prover. Delicate changes to E's term representation were needed to support partial application and especially applied variables. In contrast, the introduction of a higher-order type system had a less dramatic impact on the prover's code.

3.1 Types

For most of its history, E supported only untyped first-order logic. Cruanes implemented support for atomic types for E 2.0 [17, p. 117]. Symbols f are declared with a type signature: $\mathsf{f} : \tau_1 \times \cdots \times \tau_m \to \tau$. Atomic types are represented by integers in memory, leading to efficient type comparisons.

In λfHOL, a type signature consists of types τ, in which the function type constructor → can be nested—e.g., $(\iota \to \iota) \to \iota \to \iota$. A natural way to represent such types is to mimic their recursive structures using tagged unions. However, this leads to memory fragmentation, and a simple operation such as querying the type of a function's ith argument would require dereferencing i pointers. We prefer a flattened representation, in which a type $\tau_1 \to \cdots \to \tau_n \to \iota$ is represented by a single node labeled with → and pointing to the array $(\tau_1, \ldots, \tau_n, \iota)$. Applying $k \leq n$ arguments to a function of the above type yields a term of type $\tau_{k+1} \to \cdots \to \tau_n \to \iota$. In memory, this corresponds to skipping the first k array elements.

To speed up type comparisons, Ehoh stores all types in a shared bank and implements perfect sharing, ensuring that types that are structurally the same are represented by the same object in memory. Type equality can then be implemented as a pointer comparison, which is as efficient as E's previous integer comparison.

Since higher-order types are structurally similar to first-order terms, it would have been possible to reuse E's term data structure to store types. We experimented with this design choice but quickly abandoned the idea. E's term module relies on a signature module to provide function code to function name mapping. For each symbol, the signature stores its type. If types are represented as terms, the signature module would depend on the term module, creating a cycle. There are workarounds to make such dependencies acceptable to the C compiler, but they are inelegant and would compromise the integrity of E's source code.

3.2 Terms

In E, terms are represented as perfectly shared directed acyclic graphs [29]. Each node, or *cell*, contains 11 fields, including `f_code`, an integer that identifies the term's head symbol (if ≥ 0) or variable (if < 0); `arity`, an integer corresponding to the number of arguments passed to the head symbol; `args`, an array of size `arity` consisting of pointers to argument terms; and `binding`, which possibly stores a substitution for a variable (if `f_code` < 0) used for unification and matching.

In first-order logic, the arity of variables is always 0, and the arity of a symbol f is given by its type signature. In higher-order logic, variables may have function type and be applied, and symbols can be applied to fewer arguments than specified by their type signatures. A natural representation of λfHOL terms as tagged unions would distinguish between variables x, symbols f, and binary applications

$s\ t$. However, this scheme suffers from memory fragmentation and linear-time access, as with the representation of types, affecting performance on purely or mostly first-order problems. Instead, we propose a flattened representation, as a generalization of E's existing data structures: Allow arguments to variables, and for symbols let `arity` be the number of actual arguments, as opposed to the declared arity, and rename the field `num_args`. This approach parallels our representation of types.

A side effect of the flattened representation is that prefix subterms are not shared. For example, the terms $\mathsf{f\ a}$ and $\mathsf{f\ a\ b}$ correspond to the flattened cells $\mathsf{f}(\mathsf{a})$ and $\mathsf{f}(\mathsf{a},\mathsf{b})$. The argument subterm a is shared, but not the prefix $\mathsf{f\ a}$. Similarly, x and $x\ \mathsf{b}$ are represented by two distinct cells, $x()$ and $x(\mathsf{b})$, and there is no connection between the two occurrences of x. In particular, despite perfect sharing, their `binding` fields are unconnected, leading to inconsistencies.

A potential solution would be to systematically traverse a clause and set the `binding` fields of all cells of the form $x(\overline{s})$ whenever a variable x is bound, but this would be inefficient and inelegant. Instead, we implemented a hybrid approach: Variables are applied by an explicit application operator @, to ensure that they are always perfectly shared. Thus, $x\ \mathsf{b\ c}$ is represented by the cell $@(x,\mathsf{b},\mathsf{c})$, where x is a shared subcell. This complicates the representation somewhat (and led to subtle bugs during development), but it is graceful, since variables never occur applied in first-order terms. The main drawback of this technique is that some normalization is necessary after substitution: Whenever a variable is instantiated by a term with a symbol head, the @ symbol must be eliminated. Applying the substitution $\{x \mapsto \mathsf{f\ a}\}$ to the cell $@(x,\mathsf{b},\mathsf{c})$ must produce the cell $\mathsf{f}(\mathsf{a},\mathsf{b},\mathsf{c})$ and not $@(\mathsf{f}(\mathsf{a}),\mathsf{b},\mathsf{c})$, for consistency with other occurrences of $\mathsf{f\ a\ b\ c}$.

There is one more complication related to the `binding` field. In E, it is easy and useful to traverse a term as if a substitution has been applied, by following all set `binding` fields. In Ehoh, this is not enough, because cells must also be normalized. To avoid repeatedly creating the same normalized cells, we introduced a `binding_cache` field that connects a $@(x,\overline{s})$ cell with its substitution. However, this cache can easily become stale when the `binding` pointer is updated. To detect this situation, we store x's `binding` value in the $@(x,\overline{s})$ cell's `binding` field (which is otherwise unused). To find out whether the cache is valid, it suffices to check that the `binding` fields of x and $@(x,\overline{s})$ are equal.

3.3 Term Orders

Superposition provers rely on term orders to prune the search space. To ensure completeness of the underlying calculus, the order must be a simplification order that can be extended to a simpification order that is total on variable-free terms. The Knuth–Bendix order (KBO) and the lexicographic path order (LPO) meet this criterion. KBO is generally regarded as the more robust and efficient option for superposition, but LPO can be used to prove some problems more quickly. E implements both. In earlier work, Blanchette and colleagues have shown that only KBO can be generalized gracefully while preserving all the necessary properties for superposition [6, 14]. For this reason, we focus on KBO.

E implements the linear-time algorithm for KBO described by Löchner [28], which relies on the tupling method to store intermediate results, avoiding repeated computations. It is straightforward to generalize the algorithm to compute the graceful λfHOL version of KBO [6]. The main difference is that when comparing two terms $\mathsf{f}\ \overline{s_m}$ and $\mathsf{f}\ \overline{t_n}$, because of partial application we may now have $m \neq n$; this required changing the implementation to perform a length-lexicographic comparison of the tuples $\overline{s_m}$ and $\overline{t_n}$.

4 Unification and Matching

Syntactic unification of λfHOL terms has a definite first-order flavor. It is decidable, and most general unifiers (MGUs) are unique up to variable renaming. For example, the unification constraint $\mathsf{f}\ (y\ \mathsf{a}) \overset{?}{=} y\ (\mathsf{f}\ \mathsf{a})$ has the MGU $\{y \mapsto \mathsf{f}\}$, whereas in full higher-order logic it would admit infinitely many independent solutions of the form $\{y \mapsto \lambda x.\ \mathsf{f}\ (\mathsf{f}\ (\cdots (\mathsf{f}\ x) \cdots))\}$. Matching is a special case of unification where only the variables on the left-hand side can be instantiated.

An easy but inefficient way to implement unification and matching for λfHOL is to apply the applicative encoding (Sect. 1), perform first-order unification or matching, and decode the resulting substitution. However, the applicative encoding introduces an undesirable overhead. Instead, we propose to generalize the first-order unification and matching procedures to operate directly on λfHOL terms.

4.1 Unification

We present our unification procedure as a nondeterministic transition system, generalizing Baader and Nipkow [4]. A unification problem consists of a finite set S of unification constraints $s_i \overset{?}{=} t_i$, where s_i and t_i are of the same type. A problem is in *solved form* if it has the form $\{x_1 \overset{?}{=} t_1, \ldots, x_n \overset{?}{=} t_n\}$, where the x_i's are distinct and do not occur in the t_j's. The corresponding unifier is $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. The transition rules attempt to bring the input constraints into solved form. They can be applied in any order and eventually reach a normal form, which is either an idempotent MGU expressed in solved form or the special value $\bot$, denoting unsatisfiability of the constraints.

The first group of rules—the *positive* rules—consists of operations that focus on a single constraint and replace it with a new (possibly empty) set of constraints:

Delete	$\{t \overset{?}{=} t\} \uplus S \Longrightarrow S$
Decompose	$\{\mathsf{f}\ \overline{s_m} \overset{?}{=} \mathsf{f}\ \overline{t_m}\} \uplus S \Longrightarrow S \cup \{s_1 \overset{?}{=} t_1, \ldots, s_m \overset{?}{=} t_m\}$
DecomposeX	$\{x\ \overline{s_m} \overset{?}{=} u\ \overline{t_m}\} \uplus S \Longrightarrow S \cup \{x \overset{?}{=} u,\ s_1 \overset{?}{=} t_1, \ldots, s_m \overset{?}{=} t_m\}$ if x and u have the same type and $m > 0$
Orient	$\{\mathsf{f}\ \overline{s} \overset{?}{=} x\ \overline{t}\} \uplus S \Longrightarrow S \cup \{x\ \overline{t} \overset{?}{=} \mathsf{f}\ \overline{s}\}$
OrientXY	$\{x\ \overline{s_m} \overset{?}{=} y\ \overline{t_n}\} \uplus S \Longrightarrow S \cup \{y\ \overline{t_n} \overset{?}{=} x\ \overline{s_m}\}$ if $m > n$

Eliminate $\quad \{x \stackrel{?}{=} t\} \uplus S \Longrightarrow \{x \stackrel{?}{=} t\} \cup \{x \mapsto t\}(S) \quad$ if $x \in \mathcal{V}ar(S) \setminus \mathcal{V}ar(t)$

The Delete, Decompose, and Eliminate rules are essentially as for first-order terms. The Orient rule is generalized to allow applied variables and complemented by a new OrientXY rule. DecomposeX, also a new rule, can be seen as a variant of Decompose that analyzes applied variables; the term u may be an application.

The rules belonging to the second group—the *negative* rules—detect unsolvable constraints:

Clash $\quad \{\mathsf{f}\ \overline{s} \stackrel{?}{=} \mathsf{g}\ \overline{t}\} \uplus S \Longrightarrow \bot \quad$ if $\mathsf{f} \neq \mathsf{g}$

ClashTypeX $\quad \{x\ \overline{s_m} \stackrel{?}{=} u\ \overline{t_m}\} \uplus S \Longrightarrow \bot \quad$ if x and u have different types

ClashLenXF $\quad \{x\ \overline{s_m} \stackrel{?}{=} \mathsf{f}\ \overline{t_n}\} \uplus S \Longrightarrow \bot \quad$ if $m > n$

OccursCheck $\quad \{x \stackrel{?}{=} t\} \uplus S \Longrightarrow \bot \quad$ if $x \in \mathcal{V}ar(t)$ and $x \neq t$

The Clash and OccursCheck rules are essentially as in Baader and Nipkow. The ClashTypeX and ClashLenXF rules are variants of Clash for applied variables.

The derivations below demonstrate the computation of MGUs for the unification problems $\{\mathsf{f}\ (y\ \mathsf{a}) \stackrel{?}{=} y\ (\mathsf{f}\ \mathsf{a})\}$ and $\{x\ (z\ \mathsf{b}\ \mathsf{c}) \stackrel{?}{=} \mathsf{g}\ \mathsf{a}\ (y\ \mathsf{c})\}$:

$$
\begin{array}{ll@{\qquad}ll}
 & \{\mathsf{f}\ (y\ \mathsf{a}) \stackrel{?}{=} y\ (\mathsf{f}\ \mathsf{a})\} & & \{x\ (z\ \mathsf{b}\ \mathsf{c}) \stackrel{?}{=} \mathsf{g}\ \mathsf{a}\ (y\ \mathsf{c})\} \\
\Longrightarrow_{\mathsf{Orient}} & \{y\ (\mathsf{f}\ \mathsf{a}) \stackrel{?}{=} \mathsf{f}\ (y\ \mathsf{a})\} & \Longrightarrow_{\mathsf{DecomposeX}} & \{x \stackrel{?}{=} \mathsf{g}\ \mathsf{a},\ z\ \mathsf{b}\ \mathsf{c} \stackrel{?}{=} y\ \mathsf{c}\} \\
\Longrightarrow_{\mathsf{DecomposeX}} & \{y \stackrel{?}{=} \mathsf{f},\ \mathsf{f}\ \mathsf{a} \stackrel{?}{=} y\ \mathsf{a}\} & \Longrightarrow_{\mathsf{OrientXY}} & \{x \stackrel{?}{=} \mathsf{g}\ \mathsf{a},\ y\ \mathsf{c} \stackrel{?}{=} z\ \mathsf{b}\ \mathsf{c}\} \\
\Longrightarrow_{\mathsf{Eliminate}} & \{y \stackrel{?}{=} \mathsf{f},\ \mathsf{f}\ \mathsf{a} \stackrel{?}{=} \mathsf{f}\ \mathsf{a}\} & \Longrightarrow_{\mathsf{DecomposeX}} & \{x \stackrel{?}{=} \mathsf{g}\ \mathsf{a},\ y \stackrel{?}{=} z\ \mathsf{b},\ \mathsf{c} \stackrel{?}{=} \mathsf{c}\} \\
\Longrightarrow_{\mathsf{Delete}} & \{y \stackrel{?}{=} \mathsf{f}\} & \Longrightarrow_{\mathsf{Delete}} & \{x \stackrel{?}{=} \mathsf{g}\ \mathsf{a},\ y \stackrel{?}{=} z\ \mathsf{b}\}
\end{array}
$$

The implementation stores open constraints in a double-ended queue. Constraints are processed from the front. New constraints are added at the front if they involve complex terms that can be dealt with swiftly by Decompose or Clash, or to the back if one side is a variable. This delays instantiation of variables (with a possible increase in term size) and allows E to detect structural clashes early.

During proof search, E repeatedly needs to test a term s for unifiability not only with some other term t but also with t's subterms. Prefix optimization speeds up this test: The subterms of t are traversed in a first-order fashion; for each such subterm $\zeta\ \overline{t_n}$, at most one prefix $\zeta\ \overline{t_k}$, with $k \leq n$, is possibly unifiable with s, by virtue of their having the same arity. For first-order problems, we can only have $k = n$, since all functions are fully applied. Using this technique, Ehoh is effectively as efficient as E on first-order terms.

The transition system introduced above always terminates with a correct answer. Our proofs follow the lines of Baader and Nipkow.

In the following, the metavariable $\mathcal{R}$ is used to range over constraint sets S and the special value $\bot$. The set of all unifiers of S is denoted by $\mathcal{U}(S)$. Note that $\mathcal{U}(S \cup S') = \mathcal{U}(S) \cap \mathcal{U}(S')$. By convention, we let $\mathcal{U}(\bot) = \emptyset$. The notation $S \Longrightarrow^{!} S'$ indicates that $S \Longrightarrow^{*} S'$ and S' is a normal form (i.e., there exists no S'' such that $S' \Longrightarrow S''$). A variable x is *solved* in S if it occurs exactly once in S, in a constraint of the form $x \stackrel{?}{=} t$.

Lemma 1. *If $S \Longrightarrow \mathcal{R}$, then $\mathcal{U}(S) = \mathcal{U}(\mathcal{R})$.*

Proof. The rules Delete, $\mathsf{Decompose}$, Orient, and $\mathsf{Eliminate}$ are essentially as in Baader and Nipkow. The same arguments carry over to λfHOL. $\mathsf{OrientXY}$ trivially preserves unifiers. For $\mathsf{DecomposeX}$, the core of the argument is as follows:

$$\begin{array}{ll} & \sigma \in \mathcal{U}(\{x\ \overline{s_m} \overset{?}{=} u\ \overline{t_m}\}) \\ \text{iff} & \sigma(x\ \overline{s_m}) = \sigma(u\ \overline{t_m}) \\ \text{iff} & \sigma(x)\ \sigma(s_1)\ \ldots\ \sigma(s_m) = \sigma(u)\ \sigma(t_1)\ \ldots\ \sigma(t_m) \\ \text{iff} & \sigma(x) = \sigma(u),\ \sigma(s_1) = \sigma(t_1),\ \ldots,\ \text{and}\ \sigma(s_m) = \sigma(t_m) \\ \text{iff} & \sigma \in \mathcal{U}(\{x \overset{?}{=} u,\ s_1 \overset{?}{=} t_1, \ldots, s_m \overset{?}{=} t_m\}) \end{array}$$

The proof of the problem's unsolvability if Clash or $\mathsf{OccursCheck}$ are applicable carries over from Baader and Nipkow. For $\mathsf{ClashTypeX}$, the argument is that $\sigma(x\ \overline{s_m}) = \sigma(u\ \overline{t_m})$ is possible only if $\sigma(x) = \sigma(u)$, which requires x and u to be of the same type. Similarly, for $\mathsf{ClashLenXF}$, if $\sigma(x\ \overline{s_m}) = \sigma(\mathsf{f}\ \overline{t_n})$ with $m > n$, we must have $\sigma(x\ \overline{s_{m-n}}) = \sigma(x)\ \sigma(s_1)\ \ldots\ \sigma(s_{m-n}) = \mathsf{f}$, which is impossible. □

Lemma 2. *If S is a normal form, then S is in solved form.*

Proof. Consider an arbitrary unification constraint $s \overset{?}{=} t \in S$. We show that in all but one cases, a rule is applicable, contradicting the hypothesis that S is a normal form. In the remaining case, s is a solved variable in S.

Case $s = x$:

- Subcase $t = x$: Delete is applicable.
- Subcase $t \neq x$ and $x \in \mathcal{V}ar(t)$: $\mathsf{OccursCheck}$ is applicable.
- Subcase $t \neq x$, $x \notin \mathcal{V}ar(t)$, and $x \in \mathcal{V}ar(S \setminus \{s \overset{?}{=} t\})$: $\mathsf{Eliminate}$ is applicable.
- Subcase $t \neq x$, $x \notin \mathcal{V}ar(t)$, and $x \notin \mathcal{V}ar(S \setminus \{s \overset{?}{=} t\})$: The variable x is solved in S.

Case $s = x\ \overline{s_m}$ for $m > 0$:

- Subcase $t = \eta\ \overline{t_n}$ for $n \geq m$: $\mathsf{DecomposeX}$ or $\mathsf{ClashTypeX}$ is applicable, depending on whether x and $\eta\ \overline{t_{n-m}}$ have the same type.
- Subcase $t = y\ \overline{t_n}$ for $n < m$: $\mathsf{OrientXY}$ is applicable.
- Subcase $t = \mathsf{f}\ \overline{t_n}$ for $n < m$: $\mathsf{ClashLenXF}$ is applicable.

Case $s = \mathsf{f}\ \overline{s_m}$:

- Subcase $t = x\ \overline{t_n}$: Orient is applicable.
- Subcase $t = \mathsf{f}\ \overline{t_n}$: Due to well-typedness, $m = n$. $\mathsf{Decompose}$ is applicable.
- Subcase $t = \mathsf{g}\ \overline{t_n}$: Clash is applicable.

Since each constraint is of the form $x \overset{?}{=} t$ where x is solved in S, the problem S is in solved form. □

Lemma 3. *If the constraint set S is in solved form, then the associated substitution is an idempotent MGU of S.*

Proof. This lemma corresponds to Lemma 4.6.3 of Baader and Nipkow. Their proof carries over to λfHOL. □

Theorem 4 (Partial Correctness). *If $S \Longrightarrow^! \bot$, then S has no solutions. If $S \Longrightarrow^! S'$, then S' is in solved form and the associated substitution is an idempotent MGU of S.*

Proof. The first part follows from Lemma 1 applied transitively. The second part follows from Lemma 1 applied transitively and Lemmas 2 and 3. □

Theorem 5 (Termination). *The relation $\Longrightarrow$ is well founded.*

Proof. First, we define the auxiliary notion of weight of a term:

$$\mathcal{W}(\zeta\ \overline{s_m}) = m + 1 + \textstyle\sum_{i=1}^{m} \mathcal{W}(s_i)$$

Well-foundedness is proved by exhibiting a measure function from constraint sets to quadruples of natural numbers (n_1, n_2, n_3, n_4), where

- n_1 is the number of unsolved variables in S;
- n_2 is the sum of all term weights: $\sum_{s \overset{?}{=} t \in S} \mathcal{W}(s) + \mathcal{W}(t)$;
- n_3 is the number of right-hand side variable heads: $|\{s \overset{?}{=} x\ \bar{t} \in S\}|$;
- n_4 is the number of arguments passed to left-hand side variable heads: $\sum_{x\ \overline{s_m} \overset{?}{=} t \in S} m$.

The following table shows that the application of each positive rule lexicographically decreases the quadruple:

	n_1	n_2	n_3	n_4
Delete	$\geq$	$>$		
Decompose	$\geq$	$>$		
DecomposeX	$\geq$	$>$		
Orient	$\geq$	$=$	$>$	
OrientXY	$\geq$	$=$	$=$	$>$
Eliminate	$>$			

Eliminate, by solving one variable, decreases n_1. Delete removes a constraint from S, decreasing the weight. Decompose reduces the weight by 2. Prefix reduces the set weight by at least $m > 0$, thanks to the presence of m in the definition of $\mathcal{W}$. The negative rules, which produce the special value $\bot$, cannot contribute to an infinite $\Longrightarrow$ chain.

For example, the quadruples corresponding to the derivation starting from $\{x\ (z\ \mathsf{b}\ \mathsf{c}) \overset{?}{=} \mathsf{g}\ \mathsf{a}\ (y\ \mathsf{c})\}$ given above are $(3, 14, 0, 1) > (2, 12, 1, 2) > (2, 12, 1, 1) > (1, 10, 1, 0) > (1, 8, 1, 0)$. □

A unification algorithm for λfHOL can be derived from the above transition system, by committing to a strategy for applying the rules. This algorithm closely follows the Ehoh implementation, abstracting away from complications such as prefix optimization. We assume a flattened representation of terms; as in Ehoh, each variable stores the term it is bound to in *binding* field (Sect. 3). We also rely on a ApplySubst function, which applies the *binding* to the top-level variable. The algorithm assumes that the terms to be unified have the same type. The pseudocode is as follows:

```
function SwapNeeded(Term $s$, Term $t$) is
  return $t.head().isVar()$
        $\wedge\ (\neg\, s.head().isVar() \vee s.num\_args > t.num\_args)$

function Deref(Term $s$) is
  while $s.head().isVar() \wedge s.head().binding \neq Null$ do
    $s \leftarrow$ ApplySubst($s$, $s.head().binding$)
  return $s$

function GobblePrefix(Term $x$, Term $t$) is
  $res \leftarrow Null$
  if $x.type().args$ is suffix of $t.head().type().args$ then
    $pref\_len \leftarrow t.head().type().arity - x.type().arity$
    if $pref\_len \leq t.num\_args$ then
      $res \leftarrow$ Term($t.head()$, $t.args[1\,..\,pref\_len]$)
  return $res$

function Unify(Term $s$, Term $t$) is
  $constraints \leftarrow$ Stack()
  $constraints.push(s)$
  $constraints.push(t)$
  while $\neg\, constraints.isEmpty()$ do
    $t \leftarrow$ Deref($constraints.pop()$)
    $s \leftarrow$ Deref($constraints.pop()$)
    if $s \neq t$ then
      if SwapNeeded($s, t$) then
        $(t, s) \leftarrow (s, t)$
      if $s.head().isVar()$ then
        $x \leftarrow s.head()$
        $prefix \leftarrow$ GobblePrefix($x, t$)
        if $prefix \neq Null$ then
          $start\_idx \leftarrow prefix.num\_args + 1$
          if $x$ occurs in $prefix$ then
            return $False$
          else
            $x.binding \leftarrow prefix$
        else
          return $False$
    else if $s.head() = t.head()$ then
        $start\_idx \leftarrow 1$
    else
        return $False$
    for $i \leftarrow start\_idx$ to $t.num\_args$ do
        $constraints.push(s.args[i - start\_idx + 1])$
        $constraints.push(t.args[i])$
return $True$
```

The UNIFY function encompasses all the transition rules given above. The $s \neq t$ test in the body of UNIFY's **while** loop corresponds to Delete. The exchange triggered by the SWAPNEEDED test corresponds to Orient or OrientXY. The GOBBLEPREFIX function determines whether Prefix is applicable; if so, it finds the appropriate prefix subterm. *Null* is returned if either ClashTypeX or ClashLenXF are applicable, resulting in a failure of UNIFY. A variable is bound to a term only if it does not occur in the term, reflecting OccursCheck. The failure that arises if s and t have different head symbols corresponds to Clash. The **for** loop near the end adds the new constraints generated by Decompose or Prefix to the constraint queue. Finally, Eliminate is applied implicitly—by following variable bindings in DEREF, we incrementally apply the substitution we build.

4.2 Matching

Given two terms s and t, the matching problem consists of finding a substitution σ such that $\sigma(s) = t$. We then write that "t is an instance of s" or "s generalizes t" and call σ *generalizing substitution*. We are interested in most general generalizations (MGGs). Matching can be reduced to unification by treating variables in t as constants [4], but E implements matching as a separate, optimized procedure.

Matching can be specified abstractly as a transition system on constraints $s_i \lesssim^? t_i$ consisting of the unification rules Decompose, Prefix, Clash, ClashTypeX, ClashLenXF (with $\lesssim^?$ instead of $\stackrel{?}{=}$) and augmented with

Double $\{x \lesssim^? t,\, x \lesssim^? t'\} \uplus S \Longrightarrow \bot$ if $t \neq t'$

ClashLenXY $\{x\ \overline{s_m} \lesssim^? y\ \overline{t_n}\} \uplus S \Longrightarrow \bot$ if $x \neq y$ and $m > n$

ClashFX $\{\mathsf{f}\ \overline{s} \lesssim^? x\ \overline{t}\} \uplus S \Longrightarrow \bot$

Interestingly, the rule Delete is not sound for matching. Suppose that we are given problem $\{x \lesssim^? x, x \lesssim^? \mathsf{g}\ x\}$. Application of Delete to the first constraint would yield end state $\{x \lesssim^? \mathsf{g}\ x\}$, even though the original problem has no solution.

Matching rules are both well founded and complete. Both properties are proven analogously to unification rules.

Theorem 6 (Partial Correctness). *If $S \Longrightarrow^! \bot$, then S has no solutions. If $S \Longrightarrow^! S'$, then S' is solved form and the associated substitution is the MGG of S.*

Theorem 7 (Termination). *The relation $\Longrightarrow$ is well founded.*

The pseudocode for matching closely follows that given for unification:

```
function MATCH(Term $s$, Term $t$) is
   $constraints \leftarrow$ STACK()
   $constraints.push(s)$
   $constraints.push(t)$
   while $\neg\ constraints.isEmpty()$ do
      $t \leftarrow constraints.pop()$
      $s \leftarrow constraints.pop()$
      if $s.head().isVar()$ then
         $x \leftarrow s.head()$
         $prefix \leftarrow$ GOBBLEPREFIX$(x, t)$
         if $prefix \neq Null \wedge (x.binding = Null \vee x.binding = prefix)$ then
            $start\_idx \leftarrow prefix.num\_args + 1$
            $x.binding \leftarrow prefix$
         else
            return $False$
      else if $s.head() = t.head()$ then
         $start\_idx \leftarrow 1$
      else
         return $False$
      for $i \leftarrow start\_idx$ to $t.num\_args$ do
         $constraints.push(s.args[i - start\_idx + 1])$
         $constraints.push(t.args[i])$
   return $True$
```

The rules that are common to unification and matching are applied in the same way. The absence of the SWAPNEEDED test reflects the absence of Orient and OrientXY rules. Similarly, the absence of DEREF calls reflects the absence of Eliminate. Binding a variable to a term is now done without checking for occurrences of the former in the latter, effectively disabling OccursCheck. The algorithm checks whether a variable is already bound before binding it, corresponding to the new Double rule. Finally, GOBBLEPREFIX will return *Null* if ClashFX or ClashLenXY applies.

5 Indexing Data Structures

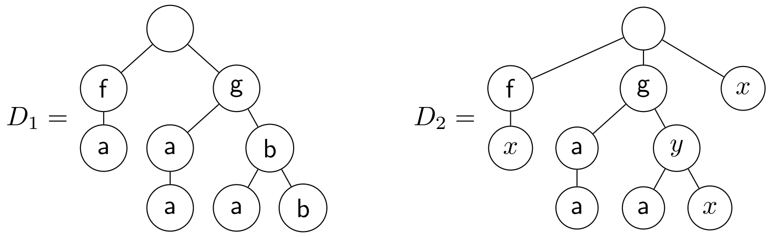
Superposition provers like E work by saturation. Their main loop heuristically selects a clause and searches for potential inference partners among a possibly large set of other clauses. Mechanisms such as simplification and subsumption also require locating terms in a large clause set. For example, when E derives a new equation $s \approx t$, if s is larger than t according to the term order, it will rewrite all instances $\sigma(s)$ of s to $\sigma(t)$ in existing clauses.

To avoid iterating over all terms (including subterms) in large clause sets, superposition provers store the potential inference partners in indexing data structures. A term index stores a set of terms S. Given a *query term* t, a query returns all terms $s \in S$ that satisfy a given *retrieval condition*: $\sigma(s) = \sigma(t)$ (s and t are unifiable), $\sigma(s) = t$ (s generalizes t), or $s = \sigma(t)$ (s is an instance of t), for some substitution σ. *Perfect* indices return exactly the subset of terms satisfying the retrieval condition. In contrast, *imperfect* indices return a superset of eligible terms, and the retrieval condition needs to be checked for each candidate.

E relies on two term indexing data structures, perfect discrimination trees [31] and fingerprint indices [40], that needed to be generalized to λfHOL. It also uses feature vector indices [41] to speed up clause subsumption and related techniques, but these require no changes to work with λfHOL clauses.

5.1 Perfect Discrimination Trees

Discrimination trees [31] are tries in which every node is labeled with a symbol or a variable. A path from the root to a leaf node corresponds to a "serialized term"—a term expressed without parentheses and commas. Consider the following discrimination trees:



Assuming $\mathsf{a}, \mathsf{b}, x, y : \iota$, $\mathsf{f} : \iota \to \iota$, and $\mathsf{g} : \iota^2 \to \iota$, the trees D_1 and D_2 represent the term sets $\{\mathsf{f}(\mathsf{a}), \mathsf{g}(\mathsf{a},\mathsf{a}), \mathsf{g}(\mathsf{b},\mathsf{a}), \mathsf{g}(\mathsf{b},\mathsf{b})\}$ and $\{\mathsf{f}(x), \mathsf{g}(\mathsf{a},\mathsf{a}), \mathsf{g}(y,\mathsf{a}), \mathsf{g}(y,x), x\}$.

E uses perfect discrimination trees for finding generalizations of query terms. For example, if the query term is $\mathsf{g}(\mathsf{a},\mathsf{a})$, it would follow the path $\mathsf{g.a.a}$ in the tree D_1 and return $\{\mathsf{g}(\mathsf{a},\mathsf{a})\}$. For D_2, it would also explore paths labeled with variables, binding them as it proceeds, and return $\{\mathsf{g}(\mathsf{a},\mathsf{a}), \mathsf{g}(y,\mathsf{a}), \mathsf{g}(y,x), x\}$.

The data structure relies on the observation that serializing is unambiguous: Assuming that symbols and variables are declared with unique types, distinct terms always give rise to distinct serialized terms. Conveniently, this property also holds for λfHOL terms. Assume that two distinct λfHOL terms yield the same serialization. Clearly, they must disagree on parentheses; one will have the subterm $s\ t\ u$ where the other has $s\ (t\ u)$. However, these two subterms cannot both be well typed; if t has the right type to be passed as argument to s, then $t\ u$ must has the wrong type.

When generalizing the data structure to λfHOL, we face a slight complication due to partial application. First-order terms can only be stored in leaf nodes, but in Ehoh we must also be able to represent partially applied terms, such as f, g, or $\mathsf{g\ a}$ (assuming, as above, that f is unary and g is binary). Conceptually, this can be solved by storing a Boolean on each node indicating whether it is an accepting state. In the implementation, the change is more subtle, because several parts of E's code implicitly assume that only leaf nodes are accepting.

The main difficulty specific to λfHOL concerns applied variables. To enumerate all generalizing terms, E needs to backtrack from child to parent nodes. To achieve this, it relies on two stacks that store subterms of the query term:

- `term_stack` stores the terms that must be matched in turn against the current subtree;
- `term_proc` stores, for each node from the root to the current subtree, the corresponding processed term, including any arguments yet to be matched.

The matching procedure starts at the root with an empty substitution σ. Initially, `term_stack` contains the query term, and `term_proc` is empty. The procedure advances by moving to a suitable child node:

A. If the node is labeled with a symbol f and the top item t of `term_stack` is of the form $\mathsf{f}(\overline{t_n})$, replace t by n new items $t_1, \ldots, t_n$, and push t onto `term_proc`.

B. If the node is labeled with a variable x, there are two subcases. If x is already bound, check that $\sigma(x) = t$; otherwise, extend σ so that $\sigma(x) = t$. Next, pop a term t from `term_stack` and push it onto `term_proc`.

The goal is to reach an accepting node. If the query term and all the terms stored in the tree are first-order, `term_stack` will then be empty, and the entire query term will have been matched.

Backtracking works in reverse: Pop a term t from `term_proc`; if the current node is labeled with an n-ary symbol, discard `term_stack`'s topmost n items; finally, push t onto `term_stack`. Variable bindings must also be undone.

As an example, looking up $\mathsf{g}(\mathsf{b}, \mathsf{a})$ in the tree D_1 would result in the following succession of stack states, starting from the root ϵ along the path $\mathsf{g.b.a}$:

	ϵ	g	$\mathsf{g.b}$	$\mathsf{g.b.a}$
`term_stack`:	$[\mathsf{g}(\mathsf{b},\mathsf{a})]$	$[\mathsf{b},\mathsf{a}]$	$[\mathsf{a}]$	$[]$
`term_proc`:	$[]$	$[\mathsf{g}(\mathsf{b},\ \mathsf{a})]$	$[\mathsf{b},\ \mathsf{g}(\mathsf{b},\ \mathsf{a})]$	$[\mathsf{a},\ \mathsf{b},\ \mathsf{g}(\mathsf{b},\ \mathsf{a})]$

(The notation $[a_1, \ldots, a_n]$ represents the n-item stack with a_1 as the top item.) Backtracking amounts to moving leftwards: When backtracking from the node g to the root, we pop $\mathsf{g}(\mathsf{b}, \mathsf{a})$ from `term_proc`, we discard two items from `term_stack`, and we push $\mathsf{g}(\mathsf{b}, \mathsf{a})$ onto `term_stack`.

To adapt the procedure to λfHOL, the key idea is that an applied variable is not very different from an applied symbol. A node labeled with an n-ary symbol or variable ζ matches a prefix t' of the k-ary term t popped from `term_stack` and leaves $n - k$ arguments $\overline{u}$ to be pushed back, with $t = t'\ \overline{u}$. If ζ is a variable, it must be bound to the prefix t'. Backtracking works analogously: Given the arity n of the node label ζ and the arity k of the term t popped from `term_proc`, we discard the topmost $n - k$ items $\overline{u}$ from `term_proc`.

To illustrate the procedure, we consider the tree D_2 but change y's type to $\iota \to \iota$. This tree represents the set $\{\mathsf{f}\ x,\ \mathsf{g\ a\ a},\ \mathsf{g}\ (y\ \mathsf{a}),\ \mathsf{g}\ (y\ x),\ x\}$. Let $\mathsf{g}\ (\mathsf{g\ a\ b})$ be the query term. We have the following sequence of substitutions and stacks:

	ϵ	g	$\mathsf{g}.y$	$\mathsf{g}.y.x$
σ:	$\emptyset$	$\emptyset$	$\{y \mapsto \mathsf{g\ a}\}$	$\{y \mapsto \mathsf{g\ a},\ x \mapsto \mathsf{b}\}$
`term_stack`:	$[\mathsf{g\ (g\ a\ b)}]$	$[\mathsf{g\ a\ b}]$	$[\mathsf{b}]$	$[]$
`term_proc`:	$[]$	$[\mathsf{g\ (g\ a\ b)}]$	$[\mathsf{g\ a\ b},\ \mathsf{g\ (g\ a\ b)}]$	$[\mathsf{b},\ \mathsf{g\ a\ b},\ \mathsf{g\ (g\ a\ b)}]$

When backtracking from $\mathsf{g}.y$ to g, by comparing y's arity of $n = 1$ with $\mathsf{g\ a\ b}$'s arity of $k = 0$, we determine that one item must be discarded from `term_stack`. Finally, to avoid traversing twice as many subterms as in the first-order case, we can optimize prefixes: Given a query term $\zeta\ \overline{t_n}$, we can also match prefixes $\zeta\ \overline{t_k}$, where $k < n$, by allowing `term_stack` to be nonempty at the end. If accepting nodes correspond to legal prefixes, the stack will then contain exactly the remaining arguments: $[t_{k+1}, \ldots, t_n]$.

Similarly to unification and matching, we present finding generalizations in a perfect discrimination tree as a transition system. States are quadruples $\mathcal{Q} = (\bar{t}, \bar{b}, D, \sigma)$, where $\bar{t}$ is a list of terms, $\bar{b}$ is a list of tuples storing information needed for backtracking, D is a discrimination (sub)tree, and σ is a substitution.

Let D be a perfect discrimination tree. $\mathcal{T}\!\mathit{erm}(D)$ denotes the set of terms stored in D. The function $D|_\zeta$ returns a singleton set containing the subtree below D's ζ edge if it exists. If D's root is accepting, $\mathit{val}(D) = (s, d)$, where s is the accepted term and d is some arbitrary data (e.g., an equation whose left- or right-hand side is t).

Starting from an initial state $([t], [], D, \emptyset)$, where t is the query term and D is an entire discrimination tree, the following transitions are possible:

AdvanceF $(\mathsf{f}\ \overline{s_m} \cdot \bar{t},\ \bar{b},\ D,\ \sigma) \rightsquigarrow (\overline{s_m} \cdot \bar{t},\ (\mathsf{f}, m, D, \sigma) \cdot \bar{b},\ D|_\mathsf{f},\ \sigma)$ if $D|_\mathsf{f}$ is defined

AdvanceX $(s\ \overline{s_m} \cdot \bar{t},\ \bar{b},\ D,\ \sigma) \rightsquigarrow (\overline{s_m} \cdot \bar{t},\ (s, m, D, \sigma) \cdot \bar{b},\ D|_x,\ \sigma[x \mapsto s])$
if $D|_x$ is defined, x and s have the same type, and $\sigma(x)$ is either undefined or equal to s

Backtrack $(\overline{s_m} \cdot \bar{t},\ (s, m, D_0, \sigma_0) \cdot \bar{b},\ D,\ \sigma) \rightsquigarrow (s\ \overline{s_m} \cdot \bar{t},\ \bar{b},\ D_0,\ \sigma_0)$

Success $([],\ \bar{b},\ D,\ \sigma) \rightsquigarrow (\mathit{val}(D),\ \sigma)$ if $\mathit{val}(D)$ is defined

In the rules, $\cdot$ denotes prepending an element or a list to a list, and $D|_\zeta$ denotes the subtree labeled by ζ, if it exists. Intuitively, AdvanceF and AdvanceX move deeper in the tree, generalizing cases A and B above to λfHOL terms. Backtrack can be used to move back to a previous state. Success rule extracts the term t and data d stored in an accepting node.

The following derivation illustrates how to locate a generalization of $\mathsf{g\ (g\ a\ b)}$ in the tree D_2:

$$
\begin{array}{ll}
 & ([\mathsf{g\ (g\ a\ b)}],\ [],\ D,\ \{\}) \\
\rightsquigarrow_{\mathsf{AdvanceF}} & ([\mathsf{g\ a\ b}],\ [(\mathsf{g}, 1, D, \emptyset)],\ D|_\mathsf{g},\ \{\}) \\
\rightsquigarrow_{\mathsf{AdvanceX}} & ([\mathsf{b}],\ [(\mathsf{g\ a}, 1, D|_\mathsf{g}, \emptyset), (\mathsf{g}, 1, D, \emptyset)],\ D|_{\mathsf{g}.y},\ \{y \mapsto \mathsf{g\ a}\}) \\
\rightsquigarrow_{\mathsf{AdvanceX}} & ([],\ [(\mathsf{b}, 0, D|_{\mathsf{g}.y}, \{y \mapsto \mathsf{g\ a}\}), (\mathsf{g\ a}, 1, D|_\mathsf{g}, \emptyset), (\mathsf{g}, 1, D, \emptyset)], \\
 & D|_{\mathsf{g}.y.x},\ \{y \mapsto \mathsf{g\ a},\ x \mapsto \mathsf{b}\}) \\
\rightsquigarrow_{\mathsf{Success}} & ((\mathsf{g}\ (y\ x), d),\ \{y \mapsto \mathsf{g\ a},\ x \mapsto \mathsf{b}\})
\end{array}
$$

Let $\rightsquigarrow_{\mathsf{Advance}} = \rightsquigarrow_{\mathsf{AdvanceF}} \cup \rightsquigarrow_{\mathsf{AdvanceX}}$. It is easy to show that Backtrack undoes an Advance transition and can be avoided when starting from an initial state.

Lemma 8. *If $\mathcal{Q} \rightsquigarrow_{\mathsf{Advance}} \mathcal{Q}'$, then $\mathcal{Q}' \rightsquigarrow_{\mathsf{Backtrack}} \mathcal{Q}$.*

Proof. There are two cases to check:

$$\begin{aligned}(\mathsf{f}\ \overline{s_m} \cdot \bar{t},\, \bar{b},\, D,\, \sigma) &\rightsquigarrow_{\mathsf{AdvanceF}} (\overline{s_m} \cdot \bar{t},\, (\mathsf{f}, m, D, \sigma) \cdot \bar{b},\, D|_{\mathsf{f}},\, \sigma)\\ &\rightsquigarrow_{\mathsf{Backtrack}} (\mathsf{f}\ \overline{s_m} \cdot \bar{t},\, \bar{b},\, D,\, \sigma)\\ (s\ \overline{s_m} \cdot \bar{t},\, \bar{b},\, D,\, \sigma) &\rightsquigarrow_{\mathsf{AdvanceX}} (\overline{s_m} \cdot \bar{t},\, (s, m, D, \sigma) \cdot \bar{b},\, D|_x,\, \sigma[x \mapsto s])\\ &\rightsquigarrow_{\mathsf{Backtrack}} (s\ \overline{s_m} \cdot \bar{t},\, \bar{b},\, D,\, \sigma)\end{aligned}$$ □

Lemma 9. *If $\mathcal{Q} \rightsquigarrow_{\mathsf{Advance}} \mathcal{Q}' \rightsquigarrow_{\mathsf{Backtrack}} \mathcal{Q}''$, then $\mathcal{Q}'' = \mathcal{Q}$.*

Proof. By Lemma 8, $\mathcal{Q}' \rightsquigarrow_{\mathsf{Backtrack}} \mathcal{Q}$. Moreover, by inspecting its defintion, we see that Backtrack is functional. Therefore, $\mathcal{Q}'' = \mathcal{Q}$. □

Lemma 10. *Let $\mathcal{Q}_0 = ([t], [], D, \emptyset)$. If $\mathcal{Q}_0 \rightsquigarrow^* \mathcal{Q}'$, then $\mathcal{Q}_0 \rightsquigarrow^*_{\mathsf{Advance}} \mathcal{Q}'$.*

Proof. Let $\mathcal{Q}_0 \rightsquigarrow \cdots \rightsquigarrow \mathcal{Q}_n = \mathcal{Q}'$. Let i be the index of the first transition $\mathcal{Q}_i \rightsquigarrow_{\mathsf{Backtrack}} \mathcal{Q}_{i+1}$. Since $\mathcal{Q}_0$'s backtracking stack is empty, we must have $i \neq 0$. Hence we have $\mathcal{Q}_{i-1} \rightsquigarrow_{\mathsf{Advance}} \mathcal{Q}_i \rightsquigarrow_{\mathsf{Backtrack}} \mathcal{Q}_{i+1}$. By Lemma 9, $\mathcal{Q}_{i-1} = \mathcal{Q}_{i+1}$. Thus, we can shorten the derivation to $\mathcal{Q}_0 \rightsquigarrow \cdots \rightsquigarrow \mathcal{Q}_{i-1} = \mathcal{Q}_{i+1} \rightsquigarrow \cdots \rightsquigarrow \mathcal{Q}_n$, thereby eliminating one Backtrack transition. By repeating this process, we can eliminate all applications of Backtrack. □

Lemma 11. *There exist no infinite chains $\mathcal{Q}_0 \rightsquigarrow_{\mathsf{Advance}} \mathcal{Q}_1 \rightsquigarrow_{\mathsf{Advance}} \cdots$.*

Proof. With each Advance transition, the height of the discrimination tree decreases by at least one. In an infinite chain, we would eventually reach a tree of height 0, consisting of a single node, from which no Advance transition is possible, contradicting the chain's existence. □

Perfect discrimination trees match a single term against a set of terms. To show that there are correct, we will connect them to the transition system $\Longrightarrow$ for matching (Sect. 4). This connection will help us show that whenever discrimination tree stores a generalization of a query term, this generalization can be found. To express the refinement argument, we introduce an intermediate transition system, $\hookrightarrow$, that focuses on a single pair of terms (like $\Longrightarrow$) but that solve the constraints in a depth-first, left-to-right fashion and build the substitution incrementally (like $\rightsquigarrow$). Its initial states are of the form $([s \precsim^? t], \emptyset)$. Its transitions are as follows:

Decompose $\quad (\mathsf{f}\ \overline{s_m} \precsim^? \mathsf{f}\ \overline{t_m} \cdot \bar{c},\, \sigma) \hookrightarrow ((s_1 \precsim^? t_1, \ldots, s_m \precsim^? t_m) \cdot \bar{c}, \sigma)$

DecomposeX $\quad (x\ \overline{s_m} \precsim^? u\ \overline{t_m} \cdot \bar{c},\, \sigma) \hookrightarrow ((s_1 \precsim^? t_1, \ldots, s_m \precsim^? t_m) \cdot \bar{c}, \sigma[x \mapsto u])$

if x and u have the same types and either $\sigma(x)$ is undefined or $\sigma(x) = u$

Success $\quad ([], \sigma) \hookrightarrow \sigma$

Clash $\quad(\mathsf{f}\ \overline{s_m} \lesssim^? \mathsf{g}\ \overline{t_n} \cdot \overline{c},\ \sigma) \hookrightarrow \bot$

$\mathsf{ClashTypeX}$ $\quad(x\ \overline{s_m} \lesssim^? u\ \overline{t_m} \cdot \overline{c},\ \sigma) \hookrightarrow \bot\quad$ if x and u have different types

$\mathsf{ClashLenXF}$ $\quad(x\ \overline{s_m} \lesssim^? \mathsf{f}\ \overline{t_n} \cdot \overline{c},\ \sigma) \hookrightarrow \bot\quad$ if $m > n$

$\mathsf{ClashLenXY}$ $\quad(x\ \overline{s_m} \lesssim^? y\ \overline{t_n} \cdot \overline{c},\ \sigma) \hookrightarrow \bot\quad$ if $x \neq y$ and $m > n$

$\mathsf{ClashFX}$ $\quad(\mathsf{f}\ \overline{s} \lesssim^? x\ \overline{t} \cdot \overline{c},\ \sigma) \hookrightarrow \bot$

Double $\quad(x\ \overline{s_m} \lesssim^? u\ \overline{t_m} \cdot \overline{c},\ \sigma) \hookrightarrow \bot$
if x and u have the same type, $\sigma(x)$ is defined, and $\sigma(x) \neq u$

We need an auxiliary function to convert $\hookrightarrow$ states to $\Longrightarrow$ states. Let

$$\alpha(\overline{c}) = \{s \lesssim^? t \mid s \lesssim^? t \in \overline{c}\} \qquad \alpha(\sigma) = \{x \lesssim^? t \mid \sigma \text{ is defined on } x \text{ and } \sigma(x) = t\}$$
$$\alpha(\overline{c}, \sigma) = \alpha(\overline{c}) \cup \alpha(\sigma) \qquad \alpha(\bot) = \bot$$

Moreover, let $\mathcal{S}$ range over states of the form $(\overline{c}, \sigma)$ and $\mathcal{R}$ additionally range over special states of the form σ or $\bot$.

Lemma 12. *If $\mathcal{S} \hookrightarrow \mathcal{R}$, then $\alpha(\mathcal{S}) \Longrightarrow^* \alpha(\mathcal{R})$.*

Proof. By case distinction on $\mathcal{R}$. Let $\mathcal{S} = (\overline{c}, \sigma)$.

Case $\mathcal{R} = (\overline{c'}, \sigma')$: The only possible rules are $\hookrightarrow_{\mathsf{Decompose}}$ and $\hookrightarrow_{\mathsf{DecomposeX}}$. If $\hookrightarrow_{\mathsf{Decompose}}$ is applied, then $\Longrightarrow_{\mathsf{Decompose}}$ is applicable and results in $\alpha(\mathcal{R})$. If $\hookrightarrow_{\mathsf{DecomposeX}}$ is applied, either we have $m > 0$, and $\Longrightarrow$ $\mathsf{DecomposeX}$ is applicable, or $m = 0$, and $\alpha(\overline{c'}, \sigma') = \alpha(\mathcal{S})$, which implies that the two states are connected by an idle transition of $\Longrightarrow^*$.

Case $\mathcal{R} = \bot$: All the $\hookrightarrow$ rules resulting in $\bot$ except for Double, have the same side conditions as the corresponding $\Longrightarrow$ rules. $\hookrightarrow_{\mathsf{Double}}$ rule corresponds to $\Longrightarrow_{\mathsf{Double}}$ rule if $m = 0$. However, if $m \neq 0$, we need an intermediate $\Longrightarrow_{\mathsf{DecomposeX}}$ step before rule $\Longrightarrow_{\mathsf{Double}}$ can be applied to derive $\bot$. Namely, since $\hookrightarrow_{\mathsf{Double}}$ is applicable, we have that $\sigma(x) = u' \neq u$. Hence, $x \lesssim^? u'$ must be present in $\alpha(\overline{c}, \sigma)$. Rule $\Longrightarrow_{\mathsf{DecomposeX}}$ will augment this set with $x \lesssim^? u$, enabling $\Longrightarrow_{\mathsf{Double}}$.

Case $\mathcal{R} = \sigma$: The only possible rule is $\hookrightarrow_{\mathsf{Success}}$, with $\overline{c} = []$. Since $\alpha(\mathcal{S}) = \alpha(\sigma)$, this transition corresponds to an idle transition of $\Longrightarrow^*$. □

Lemma 13. *If $\mathcal{S} \hookrightarrow^! \mathcal{R}$, then $\mathcal{R}$ is either some substitution σ' or $\bot$. If $\mathcal{S} \hookrightarrow^! \sigma'$, then σ' is the MGG of $\alpha(\mathcal{S})$. If $\mathcal{S} \hookrightarrow^! \bot$, then $\alpha(\mathcal{S})$ has no solutions.*

Proof. First, we show that states $\mathcal{S}' = (\overline{c'}, \sigma')$ cannot be normal forms, by exhibiting transitions from such states. If $\overline{c'} = []$, the $\hookrightarrow_{\mathsf{Success}}$ rule would apply. Otherwise, let $\overline{c'} = c_1 \cdot \overline{c''}$, and consider the matching problem $\{c_1\} \cup \alpha(\sigma')$. If this problem is in solved form, c_1 is a constraint corresponding to a solved variable, and we can apply $\hookrightarrow_{\mathsf{DecomposeX}}$ to move the constraint into the substitution. Otherwise, some $\Longrightarrow$ rule can be applied. This rule necessarily focuses on c_1, since the constraints from $\alpha(\sigma')$ correspond to solved variables. The homologous $\hookrightarrow$ rule can be applied to $\mathcal{S}'$.

Second, by Lemma 12, if $\mathcal{S} \hookrightarrow^! \sigma'$, then $\alpha(\mathcal{S}) \Longrightarrow^* \alpha(\sigma')$. By construction, $\alpha(\sigma')$ is in solved form. Thus, $\alpha(\mathcal{S}) \Longrightarrow^! \alpha(\sigma')$. By Theorem 6, the substitution corresponding to $\alpha(\sigma')$—that is, σ'—is the MGG of $\alpha(\mathcal{S})$.

Third, by Lemma 12, if $\mathcal{S} \hookrightarrow^! \bot$, then $\alpha(\mathcal{S}) \Longrightarrow^! \bot$. By Theorem 6, $\alpha(\mathcal{S})$ has no solutions. □

Lemma 14. *The relation $\hookrightarrow$ is well founded.*

By Lemma 12, every $\hookrightarrow$ transition corresponds zero or more $\Longrightarrow$ transitions. Since $\Longrightarrow$ is well founded (by Theorem 7), the only transitions that can violate well-foundedness of $\hookrightarrow$ are the ones that perform idle $\Longrightarrow$ transitions: $\hookrightarrow_{\mathsf{DecomposeX}}$ for $m = 0$ and $\hookrightarrow_{\mathsf{Success}}$. The latter is terminal, so it cannot contribute to ab infinite $\hookrightarrow$ chains. And since $\hookrightarrow_{\mathsf{DecomposeX}}$, with $m = 0$, decreases the length of $\alpha(\overline{c})$, it can be applied at most finitely many times. Thus $\hookrightarrow$ is well founded.

Lemma 15. *If term s generalizes term t, then $([s \lesssim^? t], \emptyset) \hookrightarrow^! \sigma$, where σ is the MGG of $s \lesssim^? t$.*

Proof. By Lemma 14, there exists a normal form $\mathcal{R}$ starting from $\mathcal{S} = ([s \lesssim^? t], \emptyset)$. Since $s \lesssim^? t$ is solvable, by Lemma 13, $\mathcal{R}$ must be the MGU for s and t. □

Lemma 16. *If there exists a term $s \in \mathcal{T}erm(D)$ that generalizes the query term t, then there exists a derivation $([t], [], D, \emptyset) \rightsquigarrow^! ((s, d), \sigma)$.*

Proof. By Lemma 15, we know that $(s \lesssim^? t, \emptyset) \hookrightarrow^! \sigma$ for each $s \in \mathcal{T}erm(D)$ generalizing t. This means that there exists a derivation of the form $([s \lesssim^? t], \emptyset) = \mathcal{S}_0 \hookrightarrow \mathcal{S}_1 \hookrightarrow \cdots \hookrightarrow \mathcal{S}_n \hookrightarrow \sigma$. The n first transitions must be $\mathsf{Decompose}$ or $\mathsf{DecomposeX}$, and the last transition must be $\mathsf{Success}$.

We show that there exists a derivation of the form $([t], [], D, \emptyset) = \mathcal{Q}_0 \rightsquigarrow \mathcal{Q}_1 \rightsquigarrow \cdots \rightsquigarrow \mathcal{Q}_n \rightsquigarrow ((s, d), \sigma)$, where $\mathcal{Q}_i = (\overline{t_i}, \overline{b_i}, D_i, \sigma_i)$ for each i. We define $\overline{t_i}$, D_i, and $\overline{b_i}$ as follows, for $i > 0$:

- The term list $\overline{t_i}$ consists of the right-hand sides of the constraints $\overline{c_i}$, in the same order.
- If a leaf node storing s was reached in n transitions, the serialization of s must be of the form $\zeta_1. \cdots .\zeta_n$. Take $D_i = D_{i-1}|_{\zeta_i}$.
- The backtracking information $\overline{b_i}$ is set to $(\zeta_i, m, D_{i-1}, \sigma_{i-1}) \cdot \overline{b_{i-1}}$, where $m = |\overline{t_i}| - |\overline{t_{i-1}}| + 1$.

It is easy to check that the sequence of quadruples $\mathcal{Q}_i$ forms a derivation:

- If $(\overline{c_i}, \sigma_i) \hookrightarrow_{\mathsf{Decompose}} (\overline{c_{i+1}}, \sigma_{i+1})$, then $\mathcal{Q}_i \rightsquigarrow_{\mathsf{AdvanceF}} \mathcal{Q}_{i+1}$.
- If $(\overline{c_i}, \sigma_i) \hookrightarrow_{\mathsf{DecomposeX}} (\overline{c_{i+1}}, \sigma_{i+1})$, then $\mathcal{Q}_i \rightsquigarrow_{\mathsf{AdvanceX}} \mathcal{Q}_{i+1}$.
- If $(\overline{c_n}, \sigma_n) \hookrightarrow_{\mathsf{Success}} \sigma$, then $\mathcal{Q}_n \rightsquigarrow_{\mathsf{Success}} ((s, d), \sigma)$. □

Lemma 17. *If $([t], [], D, \emptyset) \rightsquigarrow^+ ((s, d), \sigma)$, then $s \in \mathcal{T}erm(D)$ and σ is the MGG of $s \lesssim^? t$.*

Proof. Let $([t], [], D, \emptyset) = \mathcal{Q}_0 \rightsquigarrow \mathcal{Q}_1 \rightsquigarrow \cdots \rightsquigarrow \mathcal{Q}_n \rightsquigarrow ((s, d), \sigma)$ be a derivation, where $\mathcal{Q}_i = (\overline{t_i}, \overline{b_i}, D_i, \sigma_i)$ for each i. Without loss of generality, by Lemma 10, we can assume that the derivation contains no $\mathsf{Backtrack}$ transitions.

The first conjunct, $s \in \mathcal{T}erm(D)$, holds by definition for any term found from an initial state. To prove the second conjunct, we first introduce a function *preord* that defines the preorder decomposition of a list of terms:

$$\begin{aligned} preord([]) &= [] \\ preord(\zeta\ \overline{s_n} \cdot \overline{xs}) &= (\zeta, \overline{s_n} \cdot \overline{xs}) \cdot preord(\overline{s_n} \cdot \overline{xs}) \end{aligned}$$

Given a term s, $preord([s])$ gives a sequence $(\zeta_1, \overline{args_1}), \ldots, (\zeta_n, \overline{args_n})$. Since $s \in \mathcal{T}erm(D)$, the sequence $D_0, \ldots, D_n$ follows the serialization of s: $D_i = D_{i-1}|_{\zeta_i}$ for each $i > 0$.

Next, we show that there exists a derivation of the form $([s \lesssim^? t], \emptyset) = \mathcal{S}_0 \hookrightarrow \cdots \hookrightarrow \mathcal{S}_n \hookrightarrow \sigma$, where $\mathcal{S}_i = (\overline{c_i}, \sigma_i)$. We define $\overline{c_i}$, for $i > 0$, as the list of constraints whose left-hand sides are $\overline{args_i}$ and right-hand sides are $\overline{t_i}$. Both lists have the same length.

We can then check that the sequence of states $\mathcal{S}_i$ forms a derivation:

- If $\mathcal{Q}_i \rightsquigarrow_{\mathsf{AdvanceF}} \mathcal{Q}_{i+1}$, then $\mathcal{S}_i \hookrightarrow_{\mathsf{Decompose}} \mathcal{S}_{i+1}$.
- If $\mathcal{Q}_i \rightsquigarrow_{\mathsf{AdvanceX}} \mathcal{Q}_{i+1}$, then $\mathcal{S}_i \hookrightarrow_{\mathsf{DecomposeX}} \mathcal{S}_{i+1}$.
- If $\mathcal{Q}_n \rightsquigarrow_{\mathsf{Success}} \sigma$, then $\mathcal{S}_n \hookrightarrow_{\mathsf{Success}} \sigma$. □

Theorem 18 (Total Correctness). *Let D be a perfect discrimination tree and t be a term. The sets $\{s \in \mathcal{T}erm(D) \mid \exists\sigma.\ \sigma(s) = t\}$ and $\{s \mid \exists d, \sigma.\ ([t], [], D, \emptyset) \rightsquigarrow^! ((s, d), \sigma)\}$ are equal.*

Proof. This follows from Lemmas 16 and 17. □

The above theorem states that all generalizations s of a term t stored in the perfect discrimination tree can be found, but it does not exclude nondeterminism. Often, both AdvanceF and AdvanceX are applicable. To find all generalizations, we need to follow both transitions. But for some applications, it is enough to find a single generalization.

To cater for both styles, E provides iterators that store the current state of a traversal through a discrimination tree. After the iterator is initialized with the root node D and the query term t, each call to FindNextVal will move the iterator to the next node that generalizes the query term and stores a value, indicating an accepting node. After all such nodes have been traversed, the iterator is set to point to *Null*.

The pseudocode for Ehoh's updated iterator-based functions and procedures is given below. The following definitions constitute the high-level interface for iterating through values incrementally or for obtaining all values.

```
function InitIter(PDTNode $D$, Term $t$) is
  $i \leftarrow$ Iterator()
  $(i.node, i.term\_stack, i.term\_proc, i.child\_iter) \leftarrow (D, [t], [], Start)$
  return $i$
```

```
procedure FindNextVal(Iterator $i$) is
  do
    FindNextNode($i$)
  while $i.node \neq Null \wedge (\neg\, i.term\_stack.isEmpty() \vee \neg\, i.node.has\_val())$

function AllVals(PDTNode $D$, Term $t$) is
  $i \leftarrow$ InitIter($D, t$)
  $hit \leftarrow$ FindNextVal($i$)
  $res \leftarrow \emptyset$
  while $i.node \neq Null$ do
    $res \leftarrow res \cup \{i.node.val()\}$
    FindNextVal($i$)
  return $res$
```

The core logic is implemented in FindNextNode, presented below. It visits edges labeled with variables before visiting edges labeled with symbols. We assume that we can iterate through the children of a node using a function NextVarChild that, given a tree node and iterator through children, advances the iterator to the child corresponding to the next variable. (There is no need to iterate through symbols, because at most one such child should be visited.) Furthermore, we assume that the iterator can also be in the distinguished states *Start* and *End*. *Start* indicates that no child has been visited yet, whereas *End* indicates that we have visited all children.

We also rely on a function Child that returns a child corresponding to a symbol or a variable, if one exists, or *Null* otherwise. The information we need for backtracking includes the last visited child of the node and whether a variable was bound.

```
procedure BacktrackToVar(Iterator $i$) is
  forever do
    if $i.term\_proc.isEmpty()$ then
      $i.node \leftarrow Null$
      return
    else
      $(t, D, child\_iter, n, x) \leftarrow i.term\_proc.pop()$
      for $i \leftarrow 1$ to $n$ do
        $i.term\_stack.pop()$
      $i.term\_stack.push(t)$
      $i.node \leftarrow D$
      $i.child\_iter \leftarrow child\_iter$
      if $x \neq Null$ then
        $x.binding \leftarrow Null$
      if $child\_iter \neq End$ then
        return

procedure FindNextNode(Iterator $i$) is
  if $i.term\_stack.isEmpty()$ then
```

```
    BACKTRACKTOVAR($i$)
  $advanced \leftarrow False$
  while $i.node \neq Null \wedge \neg\, advanced$ do
    while $i.child\_iter \neq End \wedge \neg\, advanced$ do
      $i.child\_iter \leftarrow$ NEXTVARCHILD($i.node, i.child\_iter$)
      if $i.child\_iter \neq End$ then
        $x \leftarrow i.child\_iter.var()$
        $t \leftarrow i.term\_stack.top()$
        $prefix \leftarrow$ GOBBLEPREFIX($x, t$)
        if $prefix \neq Null \wedge (x.binding = Null \vee x.binding = prefix)$ then
          $i.term\_stack.pop()$
          $pushed \leftarrow t.num\_args - prefix.num\_args$
          for $j \leftarrow t.num\_args()$ downto $prefix.num\_args + 1$ do
            $i.term\_stack.push(t.args[j])$
          if $x.binding = Null$ then
            $x.binding \leftarrow prefix$
            $i.term\_proc.push((t, i.node, i.child\_iter, pushed, x))$
          else
            $i.term\_proc.push((t, i.node, i.child\_iter, pushed, Null))$
          $i.node \leftarrow$ CHILD($i.node, x$)
          $advanced \leftarrow True$
    $t \leftarrow i.term\_stack.top()$
    if $i.child\_iter = End$
        $\wedge \neg\, t.head().isVar() \wedge$ CHILD($D, t.head()$) $\neq Null$ then
      $i.term\_stack.pop()$
      for $j \leftarrow t.num\_args$ downto 1 do
        $i.term\_stack.push(t.args[j])$
      $i.term\_proc.push((t, i.node, i.child\_iter, t.num\_args, Null))$
      $i.node \leftarrow$ CHILD($i.node, t.head()$)
      $advanced \leftarrow True$
    if $\neg\, advanced$ then
      BACKTRACKTOVAR($i$)
    else
      $i.child\_iter \leftarrow Start$
```

5.2 Fingerprint Indices

Fingerprint indices [40] trade perfect indexing for a compact memory representation and more flexible retrieval conditions. The basic idea is to compare terms by looking only at a few predefined sample positions. If we know that term s has symbol f at the head of the subterm at 2.1 and term t has g at the same position, we can directly conclude that s and t are not unifiable.

Let A (“at a variable”), B (“below a variable”), and N (“nonexistent”) be distinguished symbols not present in the signature. Given a term t and a position p, the *fingerprint function* $\mathcal{G}\mathit{fpf}$ is defined as

$$\mathcal{G}\mathit{fpf}(t,p) = \begin{cases} \mathsf{f} & \text{if } t|_p \text{ has a symbol head } \mathsf{f} \\ \mathsf{A} & \text{if } t|_p \text{ is a variable} \\ \mathsf{B} & \text{if } t|_q \text{ is a variable for some proper prefix } q \text{ of } p \\ \mathsf{N} & \text{otherwise} \end{cases}$$

Based on a fixed tuple of sample positions $\overline{p_n}$, the *fingerprint* of a term t is defined as $\mathcal{F}\mathit{p}(t) = \big(\mathcal{G}\mathit{fpf}(t,p_1),\ldots,\mathcal{G}\mathit{fpf}(t,p_n)\big)$. To compare two terms s and t, it suffices to check that their fingerprints are componentwise compatible using the following unification and matching matrices:

	f_1	f_2	A	B	N
f_1		✗			✗
f_2	✗				✗
A					✗
B					
N	✗	✗	✗		

	f_1	f_2	A	B	N
f_1		✗	✗	✗	✗
f_2	✗		✗	✗	✗
A				✗	✗
B					
N	✗	✗	✗	✗	

The rows and columns correspond to s and t, respectively. The metavariables f_1 and f_2 represent arbitrary distinct symbols. Incompatibility is indicated by ✗.

As an example, let $(\epsilon, 1, 2, 1.1, 1.2, 2.1, 2.2)$ be the sample positions, and let $s = \mathsf{f}(\mathsf{a}, x)$ and $t = \mathsf{f}(\mathsf{g}(x), \mathsf{g}(\mathsf{a}))$ be the terms to unify. Their fingerprints are

$$\mathcal{F}\mathit{p}(s) = (\mathsf{f}, \mathsf{a}, \mathsf{A}, \mathsf{N}, \mathsf{N}, \mathsf{B}, \mathsf{B})$$
$$\mathcal{F}\mathit{p}(t) = (\mathsf{f}, \mathsf{g}, \mathsf{g}, \mathsf{A}, \mathsf{N}, \mathsf{a}, \mathsf{N})$$

Using the left matrix, we compute the compatibility vector (–, ✗, –, ✗, –, –, –). The mismatches at positions 1 and 1.1 indicate that s and t are not unifiable.

A fingerprint index is a trie that stores a term set T keyed by fingerprint. The term $\mathsf{f}(\mathsf{g}(x), \mathsf{g}(\mathsf{a}))$ above would be stored in the node addressed by $\mathsf{f.g.g.A.N.a.N}$, possibly together with other terms that share the same fingerprint. This organization makes it possible to unify or match a query term s against all the terms T in one traversal. Once a node storing the terms $U \subseteq T$ has been reached, due to overapproximation we must apply unification or matching on s and each $u \in U$.

When adapting this data structure to λfHOL, we must first choose a suitable notion of position in a term. Conventionally, higher-order positions are strings over $\{1, 2\}$ indicating, for each binary application $t_1\ t_2$, which term t_i to follow. Given that this is not graceful, it seems preferable to generalize the first-order notion to flattened λfHOL terms—e.g., $x\ \mathsf{a}\ \mathsf{b}\,|_1 = \mathsf{a}$ and $x\ \mathsf{a}\ \mathsf{b}\,|_2 = \mathsf{b}$. However, this approach fails on applied variables. For example, although $x\ \mathsf{b}$ and $\mathsf{f}\ \mathsf{a}\ \mathsf{b}$ are unifiable (using $\{x \mapsto \mathsf{f}\ \mathsf{a}\}$), sampling position 1 would yield a clash between b and a. To ensure that positions remain stable under substitution, we propose to number arguments in reverse:

$$t|^{\epsilon} = t \qquad\qquad \zeta\ t_n\ \ldots\ t_1\,|^{i.p} = t_i|^p \quad \text{if } 1 \le i \le n$$

We use a nonstandard notation, $t|^p$, for this nonstandard notion. The operation is undefined for out-of-bound indices.

Lemma 19. *Let s and t be unifiable terms, and let p be a position such that the subterms $s|^p$ and $t|^p$ are defined. Then $s|^p$ and $t|^p$ are unifiable.*

Proof. By structural induction on p.

CASE $p = \epsilon$: Trivial.

CASE $p = q.i$: Let $s|^q = \zeta\ s_m \ldots s_1$ and $t|^q = \eta\ t_n \ldots t_1$. Since p is defined in both s and t, we have $s|^p = s_i$ and $t|^p = t_i$. By the induction hypothesis, $s|^q$ and $t|^q$ are unifiable, meaning that there exists a substitution σ such that $\sigma(\zeta\ s_m \ldots s_1) = \sigma(\eta\ t_n \ldots t_1)$. Hence, $\sigma(s_1) = \sigma(t_1), \ldots, \sigma(s_i) = \sigma(t_i)$. Thus, $\sigma(s|^p) = \sigma(t|^p)$. □

Let $t\langle^p$ denote the subterm $t|^q$ such that q is the longest prefix of p for which $t|^q$ is defined. The λfHOL version of the fingerprint function is defined as follows:

$$\mathcal{G}\mathit{fpf}'(t,p) = \begin{cases} \mathsf{f} & \text{if } t|^p \text{ has a symbol head } \mathsf{f} \\ \mathsf{A} & \text{if } t|^p \text{ has a variable head} \\ \mathsf{B} & \text{if } t|^p \text{ is undefined but } t\langle^p \text{ has a variable head} \\ \mathsf{N} & \text{otherwise} \end{cases}$$

Except for the reversed numbering scheme, $\mathcal{G}\mathit{fpf}'$ coincides with $\mathcal{G}\mathit{fpf}$ on first-order terms. The fingerprint $\mathcal{F}\!p'(t)$ of a term t is defined analogously as before, and the same compatibility matrices can be used.

The most interesting new case is that of an applied variable. Given the sample positions $(\epsilon, 2, 1)$, the fingerprint of x is $(\mathsf{A}, \mathsf{B}, \mathsf{B})$ as before, whereas the fingerprint of $x\ \mathsf{c}$ is $(\mathsf{A}, \mathsf{B}, \mathsf{c})$. As another example, let $(\epsilon, 2, 1, 2.2, 2.1, 1.2, 1.1)$ be the sample positions, and let $s = x\ (\mathsf{f\ b\ c})$ and $t = \mathsf{g\ a}\ (y\ \mathsf{d})$. Their fingerprints are

$$\mathcal{F}\!p'(s) = (\mathsf{A}, \mathsf{B}, \mathsf{f}, \mathsf{B}, \mathsf{B}, \mathsf{b}, \mathsf{c})$$
$$\mathcal{F}\!p'(t) = (\mathsf{g}, \mathsf{a}, \mathsf{A}, \mathsf{N}, \mathsf{N}, \mathsf{B}, \mathsf{d})$$

The terms are not unifiable due to the incompatibility at position 1.1 (c versus d).

We can easily support prefix optimization for both terms s and t being compared: We ensure that s and t are fully applied, by adding enough fresh variables as arguments, before computing their fingerprints.

Because fingerprint indexing is an imperfect method, its correctness theorem must be stated weakly. Terms that have incompatible fingerprint cannot be unified or matched, but the converse does not hold.

Lemma 20. *If s and t are unifiable, then $\mathcal{G}\mathit{fpf}'(s,p)$ and $\mathcal{G}\mathit{fpf}'(t,p)$ are compatible according to the unification matrix.*

Proof. By contraposition, it suffices to consider the eight blank cells in the unification matrix. Since unifiability is a symmetric relation, we can further restrict our focus to four cases:

$$\begin{array}{rcllll} \mathcal{G}\mathit{fpf}'(s,p) & = & \mathsf{f}_1 & \mathsf{f}_1 & \mathsf{f}_2 & \mathsf{A} \\ \mathcal{G}\mathit{fpf}'(t,p) & = & \mathsf{f}_2 & \mathsf{N} & \mathsf{N} & \mathsf{N} \end{array}$$

The last three cases admit the same argument.

CASE $\mathsf{f}_1, \mathsf{f}_2$: By definition of $\mathcal{G}\mathit{fpf}'$, $s|^p$ and $t|^p$ must be of the forms $\mathsf{f}_1\ \overline{s}$ and $\mathsf{f}_2\ \overline{t}$, respectively. Clearly, $s|^p$ and $t|^p$ are not unifiable. By Lemma 19, s and t are not unifiable.

CASE f_1, N; f_2, N; or A, N: From $\mathcal{G}\mathit{fpf}'(s, p) = \mathsf{N}$, we deduce that $p \neq \epsilon$. Let $p = q.i.r$, where q is the longest prefix such that $\mathcal{G}\mathit{fpf}'(t, q) \neq \mathsf{N}$. Since $\mathcal{G}\mathit{fpf}'(t, q.i) = \mathsf{N}$, the head of $t|^q$ must be some symbol g. (For a variable head, we would have $\mathcal{G}\mathit{fpf}'(t, q.i) = \mathsf{B}$.) Hence, $t|^q$ has the form $\mathsf{g}\ t_n\ \ldots\ t_1$, for $n < i$. Since $q.i$ is a legal position in s, $s|^q$ has the form $\zeta\ s_m\ \ldots\ s_1$, with $i \leq m$. A necessary condition for $\sigma(s|^q) = \sigma(t|^q)$ is that $\sigma(\zeta\ s_m\ \ldots\ s_{n+1}) = \sigma(\mathsf{g})$, but this is impossible because the left-hand side is an application (since $n < m$), whereas the right-hand side is the symbol g. By Lemma 19, s and t are not unifiable. □

Lemma 21. *If s generalizes t, then $\mathcal{G}\mathit{fpf}'(s, p)$ and $\mathcal{G}\mathit{fpf}'(t, p)$ are compatible according to the matching matrix.*

Proof. The proof is by case distinction, similarly to Lemma 20. □

Theorem 22 (Overapproximation). *If s and t are unifiable, then $\mathcal{F}\mathit{p}'(s)$ and $\mathcal{F}\mathit{p}'(t)$ are compatible according to the unification matrix. If s generalizes t, then $\mathcal{F}\mathit{p}'(s)$ and $\mathcal{F}\mathit{p}'(t)$ are compatible according to the matching matrix.*

Proof. This follows from Lemmas 20 and 21.

5.3 Feature Vector Indices

Subsumption is a crucial operation to prune the search space. A clause C subsumes another clause D if there exists a subsitution σ such that $\sigma(C) \subseteq D$. Feature-vector indices [41] are an imperfect indexing data structure that can be used to retrieve clauses that subsume a query clause (forward subsumption) or that are subsumed by the query clause (backward subsumption).

Feature-vector indices are similar to fingerprint indices, but they cannot work positionally, because literals are unordered in a clause. Each clause is represented by a vector of numerical features. The features must be compatible with the subsumption relation in the following sense: For any feature f, whenever C subsumes D, we must have $f(C) \leq f(D)$. For example, the number of occurrences of a given symbol c is a legal feature.

Feature vectors are organized in a trie. When performing a lookup, we must consider all paths along which the query term's vector entries are pointwise greater (for forward subsumption) or less (for backward subsumption) than or equal to the corresponding trie nodes.

Unlike for discrimination trees and fingerprint indices, no changes were necessary to adapt feature vectors indices to λfHOL. All the predefined features make sense in λfHOL and are compatible with subsumption.

6 Inference Rules

Saturating provers try to show the unsatisfiability of a set of clauses by systematically adding logical consequences (up to simplification and redundancy), eventually deriving the empty clause as an explicit witness of unsatisfiability. They employ two kinds of inference rules: *generating rules* produce new clauses and are necessary for completeness, whereas *simplification rules* delete existing clauses or replace them by simpler clauses. This simplification is crucial for success, and most modern provers spend a large part of their time on simplification.

E implements a variant of the *given-clause* algorithm. The proof state is represented by two disjoint subsets of clauses, the *processed* clauses P and the *unprocessed* clauses U. Initially, all clauses are unprocessed. At each iteration of the main loop, the prover heuristically selects a *given clause* from U, adds it to P, and performs all generating inferences between this clause and all clauses in P. Resulting new clauses are added to U. This maintains the invariant that all direct consequences between clauses in P have been performed. Simplifiation is performed on the given clause (using clause in P as side premises), on clauses in P (using the given clause), and on newly generated clauses (again, using P).

Ehoh implements essentially the same logical calculus as E, except that it is generalized to λfHOL terms. The standard inference rules and completeness proof of superposition can be reused verbatim; the only changes concern the basic definitions of terms and substitutions [8, Sect. 1]. Completeness of superposition for λfHOL terms has been formally proved by Peltier [35] using Isabelle.

6.1 The Generating Rules

The superposition calculus consists of the following four core generating rules, whose conclusions are added to the proof state:

$$\frac{s \not\approx s' \lor C}{\sigma(C)}\,\text{ER} \qquad \frac{s \approx t \lor s' \approx u \lor C}{\sigma(t \not\approx u \lor s \approx u \lor C)}\,\text{EF}$$

$$\frac{s \approx t \lor C \qquad u[s'] \not\approx v \lor D}{\sigma(u[t] \not\approx v \lor C \lor D)}\,\text{SN} \qquad \frac{s \approx t \lor C \qquad u[s'] \approx v \lor D}{\sigma(u[t] \approx v \lor C \lor D)}\,\text{SP}$$

In each rule, σ denotes the MGU of s and s'. Not shown are order- and selection-based side conditions that restrict the rules' applicability.

Equality resolution and factoring (ER and EF) are single-premise rules that work on entire terms that occur on either side of a literal occurring in the given clause. To generalize them, it suffices to disable prefix optimization for our unification algorithm.

The rules for superposition into negative and positive literals (SN and SP) are more complex. As two-premise rules, they require the prover to find a partner for the given clause. There are two cases to consider, depending on whether the given clause acts as the first or second premise in an inference. Moreover, since the rules operate on subterms s' of a clause, it is important to be able to efficiently locate all relevant subterms, including λfHOL-specific prefix subterms.

To cover the case where the given clause acts as the left premise, the prover relies on a fingerprint index to compute a set of clauses containing terms possibly unifiable with a side s of a positive literal of the given clause. Thanks to our generalization of fingerprints, in Ehoh this candidate set is guaranteed to overapproximate the set of all possible inference partners. The unification algorithm is then applied to filter out unsuitable candidates. Thanks to prefix optimization, we can avoid gracelessly polluting the index with all prefix subterms.

For the case where the given clause is the right premise, the prover traverses its subterms s' looking for inference partners in another fingerprint index, which contains only entire left- and right-hand sides of equalities. Like E, Ehoh traverses subterms in a first-order fashion. If prefix unification succeeds, Ehoh determines the unified prefix and applies the appropriate inference instance.

6.2 The Simplifying Rules

Unlike generating rules, simplifying rules do not necessarily add conclusions to the proof state—they can also remove premises. E implements over a dozen simplifying rules, with unconditional rewriting and clause subsumption as the most significant examples. Here, we restrict our attention to a single rule, which best illustrates the challenges of supporting λfHOL:

$$\frac{s \approx t \qquad u[\sigma(s)] \approx u[\sigma(t)] \vee C}{s \approx t}\,\text{ES}$$

Given an equation $s \approx t$, equality subsumption (ES) removes a clause containing a literal whose two sides are equal except that an instance of s appears on one side where the corresponding instance of t appears on the other side.

E maintains a perfect discrimination tree that stores clauses of the form $s \approx t$ indexed by s and t. When applying the ES rule, E considers each positive literal $u \approx v$ of the given clause in turn. It starts by taking the left-hand side u as a query term. If an equation $s \approx t$ (or $t \approx s$) is found in the tree, with $\sigma(s) = u$, the prover checks whether $\sigma'(t) = v$ for some (possibly nonstrict) extension σ' of σ. If so, ES is applicable, with a second premise of the form $\sigma(s) \approx \sigma(t) \vee C$.

To consider nonempty contexts, the prover traverses the subterms u' and v' of u and v in lockstep, as long as they appear under identical contexts. Thanks to prefix optimization, when Ehoh is given a subterm u', it can find an equation $s \approx t$ in the tree such that $\sigma(s)$ is equal to some prefix of u', with n arguments $\overline{u_n}$ remaining as unmatched. Checking for equality subsumption then amounts to checking that $v' = \sigma'(t)\ \overline{u_n}$, for some extension σ' of σ.

For example, let $\mathsf{f}\ (\mathsf{g}\ \mathsf{a}\ \mathsf{b}) \approx \mathsf{f}\ (\mathsf{h}\ \mathsf{g}\ \mathsf{b})$ be the given clause, and suppose that $x\ \mathsf{a} \approx \mathsf{h}\ x$ is indexed. Under context $\mathsf{f}\ [\,]$, Ehoh considers the subterms $\mathsf{g}\ \mathsf{a}\ \mathsf{b}$ and $\mathsf{h}\ x\ \mathsf{b}$. It finds the prefix $\mathsf{g}\ \mathsf{a}$ of $\mathsf{g}\ \mathsf{a}\ \mathsf{b}$ in the tree, with $\sigma = \{x \mapsto \mathsf{g}\}$. The prefix $\mathsf{h}\ \mathsf{g}$ of $\mathsf{h}\ \mathsf{g}\ \mathsf{b}$ matches the indexed equation's right-hand side $\mathsf{h}\ x$ using the same substitution, and the remaining argument in both subterms, b, is identical. Ehoh concludes that the given clause is redundant.

7 Heuristics

E's heuristics are largely independent of the prover's logic and work unchanged for Ehoh. On first-order problems, Ehoh's behavior is virtually the same as E's. Yet, in preliminary experiments, we observed that some λfHOL benchmarks were proved quickly by E in conjunction with the applicative encoding (Sect. 1) but timed out with Ehoh. There were enough problems of this kind to prompt us to take a closer look. Based on these observations, we extended the heuristics to exploit λfHOL-specific features.

7.1 Term Order Generation

The inference rules and the redundancy criterion are parameterized by a term order—typically an instance of KBO or LPO (Sect. 3). E can generate a *symbol weight* function (for KBO) and a *symbol precedence* (for KBO and LPO) based on criteria such as the symbols' frequencies, their arities, and whether they appear in the conjecture.

In preliminary experiments, we discovered that the presence of an explicit application operator @ can be beneficial for some problems. A small example will help illustrate this behavior. Let $\mathsf{a} : \iota_1$, $\mathsf{b} : \iota_2$, $\mathsf{c} : \iota_3$, $\mathsf{f} : \iota_1 \to \iota_2 \to \iota_3$, $x : \iota_2 \to \iota_3$, $y : \iota_2$, and $z : \iota_3$, and consider the clauses

$$\mathsf{f}\ \mathsf{a}\ y \not\approx \mathsf{c} \qquad\qquad x\ \mathsf{b} \approx z$$

where the first one is the negated conjecture. Their applicative encoding is

$$@_{\iota_2,\iota_3}(@_{\iota_1,\iota_2\to\iota_3}(\mathsf{f},\mathsf{a}),y) \not\approx \mathsf{c} \qquad\qquad @_{\iota_2,\iota_3}(x,\mathsf{b}) \approx z$$

where $@_{\tau,\upsilon}$ is a type-indexed family of explicit application symbols representing the application of a function of type $\tau \to \upsilon$. With the applicative encoding, generation schemes can take the symbols $@_{\tau,\upsilon}$ into account, effectively exploiting the type information carried by such symbols. Since $@_{\iota_2,\iota_3}$ is a conjecture symbol, some weight generation scheme could give it a low weight, which would also impact the second clause. By contrast, the native λfHOL clauses share no symbols; the connection between them is hidden in the types of variables and symbols, which are ignored by the heuristics.

To simulate the desirable behavior observed on applicatively encoded problems, we introduced four generation schemes that extend E's existing symbol-frequency-based schemes by partitioning the symbols by type. To each symbol, the new schemes assign a frequency corresponding to the sum of all symbol frequencies for its class. Each new scheme is inspired by a similarly named type-agnostic scheme in E, without `type` in its name:

- `typefreqcount` assigns as each symbol's weight the number of occurrences of symbols that have the same type.
- `typefreqrank` sorts the frequencies obtained using `typefreqcount` in increasing order and assigns each symbol a weight corresponding to its rank.

- `invtypefreqcount` is `typefreqcount`'s inverse. Whenever `typefreqcount` would assign a weight w to a symbol, it assigns $M - w + 1$, where M is the maximum symbol weight according to `typefreqcount`.
- `invtypefreqrank` is `typefreqrank`'s inverse. It sorts the frequencies in decreasing order.

In addition, we designed four schemes that combine E's type-agnostic and Ehoh's type-aware approaches using a linear equation:

- `combfreqcount` assigns as each symbol's weight the value $2w_1 + w_2$, where w_1 is the weight according to `freqcount` (i.e., the symbol's frequency) and w_2 is the weight according to `typefreqcount`.
- `combfreqrank` sorts the frequencies obtained using `combfreqcount` in increasing order and assigns each symbol a weight corresponding to its rank.
- `invcombfreqcount` is `combfreqcount`'s inverse. Whenever `combfreqcount` would assign a weight w to a symbol, it assigns $M - w + 1$, where M is the maximum symbol weight according to `combfreqcount`.
- `invcombfreqrank` is `combfreqrank`'s inverse. It sorts the frequencies in decreasing order.

To generate symbol precedences, E can sort symbols by weight and use the symbol's position in the sorted array as the basis for precedence. To account for the type information introduced by the applicative encoding, we implemented four type-aware precedence generation schemes, called `typefreq`, `invtypefreq`, `combfreq`, and `invcombfreq`, that sort the symbols by weight according to `typefreqcount`, `invtypefreqcount`, `combfreqcount`, and `invcombfreqcount`, respectively. Ties are broken by comparing the symbols' number of occurrences and, if necessary, the position of their first occurrence in the input problem.

7.2 Literal Selection

The side conditions of the superposition rules (SN and SP, Sect. 6.1) allow the use of a literal selection function to restrict the set of *inference literals*, thereby pruning the search space. Given a clause, a literal selection function returns a (possibly empty) subset of its literals. For completeness, any nonempty subset selected must contain at least one negative literal. If no literal is selected, all *maximal* literals become inference literals. The most widely used function in E is probably `SelectMaxLComplexAvoidPosPred`, which we abbreviate to `SelectMLCAPP`. It selects at most one negative literal, based on size, groundness, and maximality of the literal in the clause. It also avoids negative literals that share a predicate symbol with a positive literal in the same clause.

Intuitively, applied variables can potentially be unified with more terms than terms with rigid heads. This makes them prolific in terms of possible inference partners, a behavior we might want to avoid. However, shorter proofs might be found if we prefer selecting applied variables. To cover both scenarios, we implemented selection functions that prefer or defer selecting applied variables.

Let $\mathit{max}(L) = 1$ if L is a maximal literal of the clause it appears in; otherwise, $\mathit{max}(L) = 0$. Let $\mathit{appvar}(L) = 1$ if L is a literal where either side is an applied variable; otherwise, $\mathit{appvar}(L) = 0$. Based on these definitions, we devised the following selection functions, both of which rely on `SelectMLCAPP` to break ties:

- `SelectMLCAPPAvoidAppVar` selects a negative literal L with the maximal value of $(\mathit{max}(L), 1 - \mathit{appvar}(L))$ according to the lexicographic order.
- `SelectMLCAPPPreferAppVar` selects a negative literal L with the maximal value of $(\mathit{max}(L), \mathit{appvar}(L))$ according to the lexicographic order.

The presence of $\mathit{max}(L)$ as the first criterion is motivated by initial experiments.

7.3 Clause Selection

Selection of the given clause is a critical choice point. E heuristically assigns *clause priorities* and *clause weights* to the candidates. The priorities provide a crude partition, whereas the weights are used to further distinguish candidates. E's main loop visits, in round-robin fashion, a set of priority queues. From each queue, it selects a number of clauses with the highest priorities, breaking ties by preferring smaller weights. Typically, one of the queues will use the clauses' age as priority, to ensure fairness.

E provides template weight functions that allow users to fine-tune parameters such as weights assigned to variables or function symbols. The most widely used template is `ConjectureRelativeSymbolWeight`, which we abbreviate to `CRSWeight`. It computes term and clause weights according to the following parameters:

conj_mul multiplier applied to the weight of conjecture symbols;
fweight weight of a nonnullary function symbol;
cweight weight of a nullary function symbol;
pweight weight of a predicate symbol;
vweight weight of a variable;
maxt_mul multiplier applied to the weight of maximal terms in a literal;
maxl_mul multiplier applied to the weight of maximal literals in a clause;
pos_mul multiplier applied to the weight of positive literals in a clause.

This templates works well for some applicatively encoded problems. To see why, let $\mathsf{a} : \iota$, $\mathsf{f} : \iota \to \iota$, $x : \iota$, and $y : \iota \to \iota$, and consider the clauses

$$y\ x \not\approx x \qquad\qquad \mathsf{f}\ \mathsf{a} \approx \mathsf{a}$$

where the first one is the negated conjecture. Their applicative encoding is

$$@_{\iota,\iota}(y, x) \not\approx x \qquad\qquad @_{\iota,\iota}(\mathsf{f}, \mathsf{a}) \approx \mathsf{a}$$

The encoded clauses share the symbol $@_{\iota,\iota}$, whose weight will be multiplied by *conj_mul*—typically a factor in the interval $(0, 1)$. By contrast, the native λfHOL clauses share no symbols, and the heuristic would fail to notice that f and y have the same type, giving a higher weight to the second clause.

To mitigate this effect, we implemented a new type-aware template function, called `CRSTypeWeight`, that behaves like `CRSWeight` except that it applies the *conj_mul* multiplier to all symbols whose type occurs in the conjecture. For the example above, since $\iota \to \iota$ appears in the conjecture, it would notice the relation between the conjecture variable y and the symbol f and multiply f's weight by *conj_mul*.

A benefit of natively supporting λfHOL is that the prover can recognize applied variables. It may make sense to penalize clauses that depend on this higher-order feature—or perhaps such clauses should be promoted instead. To exploit this kind of information in heuristics, we extended most of E's weight function templates, as well as `CRSTypeWeight`, with the following optional parameter:

appv_mul multiplier applied to terms that constitute either side of a literal and whose head is a variable applied to at least one argument.

In addition, we implemented a new clause priority scheme, `ByAppVarNum`, that separates the clauses by the number of top-level applied variables occurring in the clause, favoring those containing fewer such variables.

7.4 Configurations and Modes

A combination of proof search parameters—including term order, literal selection, and clause selection—is called a *configuration*. It is generally acknowledged that most provable problems arising in practice can be proved quickly in at least one configuration; but the configuration space being infinite, there is no easy way to find this configuration.

For years, E has provided an *auto* mode, which analyzes the input problem and chooses a configuration known to perform well on similar problems. More recently, E has been extended with an *autoschedule* mode, which applies a portfolio of configurations in sequence on the given problem, restarting the prover for each configuration. Time slicing approaches tend to perform better in practice, even if each configuration is given a shorter time slice.

Configurations that perform well on a wide range of problems have emerged over time. One of them is the configuration that is most often chosen by E's *auto* mode. We call it *boa* ("best of *auto*"):

Term order:	KBO
Weight generation:	`invfreqrank`
Precedence generation:	`invfreq`
Literal selection:	`SelectMLCAPP`

Clause selection:	`1.CRSWeight(SimulateSOS,`
	`0.5, 100, 100, 100, 100, 1.5, 1.5, 1),`
	`4.CRSWeight(ConstPrio,`
	`0.1, 100, 100, 100, 100, 1.5, 1.5, 1.5),`

```
1.FIFOWeight(PreferProcessed),
1.CRSWeight(PreferNonGoals,
    0.5, 100, 100, 100, 100, 1.5, 1.5, 1),
4.Refinedweight(SimulateSOS, 3, 2, 2, 1.5, 2)
```

The clause selection scheme consists of five queues, each of which is specified by a weight function template. The prefixes n. next to the template names indicate that n clauses should be taken from the corresponding queue each time it is visited. The first argument to each template is the clause priority scheme.

8 Preprocessing

E includes a preprocessor that transforms first-order formulas into clausal normal form, before the main loop is started and inference rules are applied. Beyond turning the problem into a conjunction of disjunctive clauses, the preprocessor eliminates quantifiers, introducing Skolem symbols for essentially existential quantifiers. For first-order logic, skolemization preserves satisfiability (unprovability) and unsatisfiability (provability). In contrast, for higher-order logics without the axiom of choice, naive skolemization is unsound, because it introduces symbols that can be used to instantiate higher-order variables.

One solution proposed by Miller [33, Sect. 6] is to ensure that Skolem symbols are always applied to a minimum number of arguments. Accordingly, the version of λfHOL introduced by Bentkamp et al. [8] distinguishes between mandatory and optional arguments. However, to keep the implementation simple, we have decided to ignore this issue and consider all arguments as optional, including those to Skolem symbols. We expect to extend Ehoh's logic to full higher-order logic with the axiom of choice, which will address the issue.

There is another transformation performed by preprocessing that is problematic, but for a different reason. *Definition unfolding* is the process of replacing equationally defined symbols with their definitions and removing the defining equations. A definition is a clause of the form $\mathsf{f}\ \overline{x_m} \approx t$, where the variables $\overline{x_m}$ are distinct, f does not occur in the right-hand side t, and $\mathcal{V}ar(t) \subseteq \{x_1, \ldots, x_m\}$. This transformation preserves unsatisfiability (provability) for first- and higher-order logic, but not for λfHOL, making Ehoh incomplete. The reason is that by removing the definitional clause, we also remove a symbol f that otherwise could be used to instantiate a higher-order quantifier. For example, the clause set $\{\mathsf{f}\ x \approx x,\ \mathsf{f}\ (y\ \mathsf{a}) \not\approx \mathsf{a}\}$ is unsatisfiable, whereas $\{y\ \mathsf{a} \not\approx \mathsf{a}\}$ is satisfiable in λfHOL. (In full higher-order logic, the second clause set would be unsatisfiable thanks to the $\{x \mapsto \lambda x.\ x\}$ instance and the β-rule.) For the moment, we have simply disabled definition unfolding in Ehoh. We will enable it again once we have added support for λ-terms.

9 Evaluation

In this section, we consider the following questions: How useful are Ehoh's new heuristics? And how does Ehoh perform compared with the previous version of E, 2.2, used directly or in conjunction with the applicative encoding, and compared with other provers? To answer the first question, we evaluated each new parameter independently. From the empirical results, we derived a new configuration optimized for λfHOL problems. To answer the second question, we compared Ehoh's success rate and speed on λfHOL problems with native higher-order provers and with E's on their applicatively encoded counterparts. We also included first-order benchmarks to measure Ehoh's overhead with respect to E.

We set a CPU time limit of 60 s per problem. This is more than allotted by interactive proof tools such as Sledgehammer, or by cooperative provers such as Leo-III and Satallax, but less than the 300 s of CASC [49]. The experiments were performed on StarExec [45] nodes equipped with Intel Xeon E5-2609 0 CPUs clocked at 2.40 GHz and with 8192 MB of memory. Our raw experimental data is publicly available.[2]

9.1 Heuristics Tuning

We used the *boa* configuration as the basis to evaluate the new heuristic schemes. For each heuristic parameter we tuned, we changed only its value while keeping the other parameters the same as for *boa*. This gives an idea of how each parameter value affects overall performance. All heuristic parameters were tested on a 5012 problem suite generated using Sledgehammer, consisting of four versions of the Judgment Day [15] suite. The problems were given in native λfHOL syntax.

Term Order Generation. Evaluating new weight and precedence generation heuristics amounts to testing each possible combination of frequency based schemes, including E's original type-agnostic schemes. Figure 1 shows the number of solved (proved or disproved) problems for each combination. In this and the next figures, the slanted number corresponds to *boa*, whereas bold singles out the best value.

Figure 1 indicates that including type information in the generation schemes results in a somewhat higher number of solved problems compared with E's type-agnostic schemes. Against our expectations, Ehoh's combined schemes are generally less efficient than its type-aware schemes.

Literal Selection. The literal selection function has little impact on performance: Ehoh solves 2379 problems with `SelectMLCAPP` or `SelectMLCAPPAvoidAppVar`, and 2378 problems with `SelectMLCAPPPreferAppVar`.

Clause Selection. Clause selection is the heuristic component that we extended the most. We must assess the effect of a new heuristic weight function, a multiplier for the occurrence of top-level applied variables, and clause priority based on the number of top-level applied variables.

[2] `http://matryoshka.gforge.inria.fr/pubs/ehoh_results.tar.gz`

	`freq`	`invfreq`	`typefreq`	`invtypefreq`	`combfreq`	`invcombfreq`
`freqcount`	2294	2288	2287	2297	2290	2287
`invfreqcount`	2371	2373	2374	2370	2369	2377
`freqrank`	2326	2317	2323	2329	2322	2318
`invfreqrank`	2383	*2379*	2376	2380	2381	2381
`typefreqcount`	2305	2314	2301	2306	2302	2311
`invtypefreqcount`	2386	2381	2389	2388	2384	2379
`typefreqrank`	2326	2334	2322	2334	2321	2336
`invtypefreqrank`	2390	2382	2390	**2394**	2387	2386
`combfreqcount`	2273	2281	2271	2285	2269	2280
`invcombfreqcount`	2380	2375	2382	2379	2380	2375
`combfreqrank`	2321	2313	2319	2321	2318	2312
`invcombfreqrank`	2368	2378	2371	2378	2368	2380

Fig. 1. Evaluation of weight and precedence generation schemes

	0.25	0.35	0.5	0.7	1	1.41	2	2.82	4
`CRSWeight`	2311	2341	2363	2374	***2379***	2376	2377	2376	2377
`CRSTypeWeight`	2331	2331	2360	2371	2372	**2374**	2373	2373	2372

Fig. 2. Evaluation of weight function and applied variable multipliers

To test the effect of the new type-based heuristic weight function, we changed *boa*'s clause selection heuristic by using `4.CRSTypeWeight(...)` where *boa* specifies `4.CRSWeight(...)`. We chose nine values between 0.25 and 4 for testing the effect of the applied variable multiplier.

Figure 2 summarizes the results of combining weight functions `CRSWeight` and `CRSTypeWeight` will the different values for the applied variable multiplier. Applying a multiplier smaller than 1, which corresponds to preferring literals containing applied variables, can result in losing dozens of solutions. Overall, it would seem that using the type-aware heuristic is slightly detrimental.

Finally, we evaluated the new clause priority `ByAppVarNum`, based on the number of top-level applied variables, by replacing `4.CRSWeight(ConstPrio,...)` with `4.CRSWeight(ByAppVarNum,...)` in *boa*'s specification. `ConstPrio`, which assigns each clause the same priority, enabled Ehoh to solve 2379 problems. By contrast, `ByAppVarNum` led to 2377 solved problems. These results suggest that `ByAppVarNum` is not particularly helpful.

New Configuration. The results presented above give an overview of how each parameter influences performance. We also evaluated their performance in combination, to derive an alternative to *boa* for λfHOL.

For each category of parameters, we chose either *boa*'s value of the parameter in *boa* ("old") or the best performing newly implemented parameter ("new"). Based on the results above, for term orders, we chose the combination of `invtypefreqrank` and `invtypefreq`; for clause selection, we chose `CRSTypeWeight` with `ConstPrio` priority and an *appv_mult* factor of 1.41; for literal selection, we chose `SelectMLCAPPAvoidAppVar`.

Term order	Literal selection	Clause weight	Solved
Old	Old	Old	*2379*
Old	Old	New	2374
Old	New	Old	2379
Old	New	New	2373
New	Old	Old	2394
New	Old	New	**2397**
New	New	Old	2395
New	New	New	**2397**

Fig. 3. Evaluation of combinations of new parameters

Figure 3 shows the number of solved problems for all combinations of these parameters. From the two configurations that solve 2397 problems, we selected the "New Old New" combination as our suggested "higher-order best of *auto*," or *hoboa*, configuration. In the next subsection, we present a more detailed evaluation of *hoboa*, along with other configurations, on a larger benchmark suite.

9.2 Main Evaluation

The benchmark problems are partitioned as follows: (1) 1147 first-order TPTP [47] problems belonging to the FOF (untyped) and TF0 (monomorphic) categories, excluding arithmetic; (2) 5012 Sledgehammer-generated problems from the Judgment Day [15] suite, targeting the monomorphic first-order logic embodied by TPTP TF0; (3) all 530 monomorphic higher-order problems from the TH0 category of the TPTP library belonging to the λfHOL fragment; (4) 5012 Sledgehammer-generated problems from Judgment Day, targeting the λfHOL fragment of TPTP TH0.

The TPTP library includes benchmarks from various fields of computer science and mathematics. It is the de facto standard for evaluating and testing automatic provers, but it has few higher-order problems. For the first group of benchmarks, we randomly chose 1000 FOF problems (out of 8172) and all monomorphic TFF problems that are parsable by E within 60 s (amounting to 147 out of 231 monomorphic TFF problems). Both groups of Sledgehammer problems include two subgroups of 2506 problems, generated to include 32 or 512 Isabelle lemmas (SH 32 and SH 512), to represent both smaller and larger problems arising in interactive verification. Each subgroup itself consists of two sub-subgroups of 1253 problems, generated by using either λ-lifting or SK-style combinators to encode λ-expressions.

We evaluated Ehoh against two higher-order provers, Leo-III and Satallax, and a version of E, which we call @+E, that first performs the applicative encoding. Leo-III and Satallax have the advantage that they can instantiate higher-order variables by λ-terms; thus, some formulas that are provable by these two systems may be nontheorems for @+E and Ehoh, or they may require reasoning about λ-lifted functions or SK-style combinators. We also evaluated E, @+E,

	First-order			Higher-order		
	TPTP	SH 32	SH 512	TPTP	SH 32	SH 512
E a	598	939	1234			
E as	**645**	950	**1311**			
E b	546	944	1243			
@+E a	526	943	1114	395	962	1119
@+E as	567	950	1151	397	965	1155
@+E b	538	942	1228	397	960	1272
Ehoh a	599	938	1233	396	962	1240
Ehoh as	644	949	1310	395	**973**	**1325**
Ehoh b	547	944	1243	396	966	1244
Ehoh hb	502	944	1231	393	968	1262
Leo-III	542	**951**	1126	**421**	963	1145
Satallax				406	768	790

Fig. 4. Number of proved problems

Ehoh, and Leo-III on first-order benchmarks to measure the overhead introduced by our extensions, as well as that entailed by the applicative encoding. (Satallax is not included because it can only parse THF problems.) The number of problems each system proved is given in Figure 4. We considered the E modes *auto* (a) and *autoschedule* (as) and the configurations *boa* (b) and *hoboa* (hb). We observe the following:

- Comparing the Ehoh rows with the corresponding E rows, we see that Ehoh's overhead is barely noticeable—the difference is at most one problem. These problems are typically solved immediately before the time limit.
- Ehoh generally outperforms the applicative encoding, on both first-order and higher-order problems. On Sledgehammer benchmarks, the best Ehoh mode (*autoschedule*) clearly outperforms all @+E modes and configurations. Despite this, there are problems that @+E proves faster than Ehoh, because the applicative encoding impacts the heuristics in subtle ways.
- Especially on large benchmarks, the E variants are substantially more successful than Leo-III and Satallax. This corroborates older experiments by Sultana, Blanchette, and Paulson [46] involving LEO-II and Satallax.
- The *hoboa* configuration outperforms *boa* on higher-order problems, suggesting that it could be worthwhile to re-train *auto* and *autoschedule* based on λfHOL benchmarks, and to design further heuristics.

Next to the number of problems proved, the time in which a prover gives an answer is also an important consideration. Figure 5 compares the average running times of systems on the problems that all of the applicable systems proved. The results show that Ehoh incurs little overhead on first-order problems. We conjectured that the native treatment of λfHOL terms, which are roughly half the size of applicatively encoded terms, would result in a factor-of-2 speed-up of Ehoh over @+E, but this is not confirmed by the evaluation.

	First-order			Higher-order		
	TPTP	SH 32	SH 512	TPTP	SH 32	SH 512
E a	0.42	0.21	0.66			
E as	0.45	0.31	1.20			
E b	0.60	**0.10**	**0.60**			
@+E a	**0.39**	0.15	0.62	0.06	0.10	0.39
@+E as	0.85	0.15	**0.60**	0.06	0.10	**0.37**
@+E b	0.51	0.14	1.00	0.06	0.09	0.72
Ehoh a	0.43	0.25	0.70	0.08	0.08	0.63
Ehoh as	0.86	0.31	1.20	0.23	0.17	1.12
Ehoh b	0.64	0.11	0.64	**0.05**	**0.07**	0.47
Ehoh hb	0.93	0.19	1.13	**0.05**	0.13	0.61
Leo-III	10.18	8.75	27.35	5.09	11.63	35.06
Satallax				2.92	5.87	9.95

Fig. 5. Average running times (s) on the problems proved by all provers

10 Discussion and Related Work

Our working hypothesis is that it should be possible to extend existing first-order provers to higher-order logic, without slowing them down unduly. Our research program is two-pronged: On the theoretical side, we are investigating higher-order extensions of superposition; on the practical side, we are implementing such extensions in a state-of-the-art prover. The work described in this report required modifying many parts of the E prover. The invariant that variables are unapplied and that symbols are always passed the same number of arguments were entrenched in E's algorithms and data structures, requiring hundreds of modifications. Nonetheless, we found the generalization manageable and are now in a position to add support for λ-terms and higher-order unification.

Most higher-order provers were developed from the ground up. Two exceptions are Otter-λ by Beeson [7] and Zipperposition by Cruanes [18]. Otter-λ adds λ-terms and second-order unification to the superposition-based Otter [30]. The approach is pragmatic, with little emphasis on completeness. Zipperposition is a superposition-based prover written in OCaml. It was initially designed for first-order logic but subsequently extended to higher-order logic. Its performance is a far cry from E's, but it is easier to modify. It competed at the 2017 edition of CASC [49] and is used by Bentkamp et al. [8] for prototyping. Finally, there is noteworthy preliminary work by the developers of Vampire [12] and of the SMT (satisfiability modulo theories) solvers CVC4 and veriT [5].

Native higher-order reasoning was pioneered by Robinson [37], Andrews [1], and Huet [23]. Andrews [2] and Benzmüller and Miller [10] provide excellent surveys. TPS, by Andrews et al. [3], was based on expansion proofs and let users specify proof outlines. The Leo family of systems, developed by Benzmüller and his colleagues, is based on resolution and paramodulation. LEO [9] featured support for extensionality on the calculus level and introduced the cooperative paradigm to integrate first-order provers. Leo-III [44] expands the cooperation with SMT (satisfiability modulo theories) solvers and introduces term orders in a pragmatic, incomplete way. Brown's Satallax [16] is based on a complete higher-order tableau calculus, guided by a SAT solver; recent versions also cooperate with first-order provers. Satallax usually wins in CASC's higher-order theorem division [49]. Another competitive prover is Lindblad's AgsyHOL [27]. It is based on a focused sequent calculus driven by a generic narrowing engine.

An alternative to all of the above is to reduce higher-order logic to first-order logic by means of a translation. Robinson [38] outlined this approach decades before tools such as Miz$\mathbb{AR}$ [50], Sledgehammer [34], HOLyHammer [24], and CoqHammer [19] popularized it in proof assistants. In addition to performing an applicative encoding, such translations must eliminate the λ-expressions [20, 32] and encode the type information [13]. In practice, on problems with a large first-order component, translations perform surprisingly well compared with the existing native provers [46]. Largely thanks to Sledgehammer, Isabelle often came in close second at CASC, even defeating Satallax in 2012 [48].

By removing the need for the applicative encoding, our current work reduces the translation gap. The encoding buries the λfHOL terms' heads under layers of @ symbols, which impacts the heuristics that inspect terms. Terms double in size, cluttering the data structures, and twice as many subterm positions must be considered for inferences. In addition, the encoding must be undone in the proofs, which can be difficult. But our main objection is that the applicative encoding is incompatible with interpreted operators, notably for arithmetic. The traditional solution is to introduce proxies to connect an uninterpreted nullary symbol with its interpreted counterpart(e.g., $@(@(\mathsf{plus}, x), y) \approx x + y$), but this is clumsy. Moreover, in a monomorphic logic, @ is not a single symbol but a type-indexed family of symbols $@_{\tau,\upsilon}$, which must be correctly introduced and recognized. While it should be possible to base a higher-order prover on such an encoding, the prospect is aesthetically and technically unappealing, and performance would likely suffer.

11 Conclusion

Despite considerable progress since the 1970s, higher-order automated reasoning has not yet assimilated some of the most successful methods for first-order logic with equality, such as superposition. We presented a graceful extension of a state-of-the-art first-order theorem prover to a fragment of higher-order logic devoid of λ-terms. Our work covers both theoretical and practical aspects. Experiments show promising results on higher-order problems and almost no overhead for first-order problems, as we would expect from a graceful generalization.

The resulting Ehoh prover will form the basis of our work towards strong higher-order automation. Our aim is to turn it into a prover that excels on proof obligations emerging from interactive verification. In our experience, these tend to be large but only mildly higher-order. Most of our techniques, including prefix optimization, are applicable to other superposition-based provers, such as SPASS and Vampire, and to the paramodulation-based higher-order Leo-III.

Acknowledgment. We are grateful to the maintainers of StarExec for letting us use their service. We thank Ahmed Bhayat, Alexander Bentkamp, Daniel El Ouraoui, Michael Färber, Pascal Fontaine, Predrag Janičić, Robert Lewis, Tomer Libal, Giles Reger, Hans-Jörg Schurr, Alexander Steen, Mark Summerfield, and Dmitriy Traytel for suggesting many improvements to this text and to Vukmirović's MSc thesis. We also want to thank the other members of the Matryoshka team, including Sophie Tourret and Uwe Waldmann, as well as Christoph Benzmüller, Andrei Voronkov, Daniel Wand, and Christoph Weidenbach, for many stimulating discussions.

Vukmirović and Blanchette's research has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka).

References

[1] Andrews, P.B.: Resolution in type theory. J. Symb. Log. 36(3), 414–432 (1971)

[2] Andrews, P.B.: Classical type theory. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 2, pp. 965–1007. Elsevier and MIT Press (2001)

[3] Andrews, P.B., Bishop, M., Issar, S., Nesmith, D., Pfenning, F., Xi, H.: TPS: A theorem-proving system for classical type theory. J. Autom. Reason. 16(3), 321–353 (1996)

[4] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)

[5] Barbosa, H., Reynolds, A., Fontaine, P., Ouraoui, D.E., Tinelli, C.: Higher-order SMT solving. In: Dimitrova, R., D'Silva, V. (eds.) SMT 2018 (2018)

[6] Becker, H., Blanchette, J.C., Waldmann, U., Wand, D.: A transfinite Knuth–Bendix order for lambda-free higher-order terms. In: de Moura, L. (ed.) CADE-26. LNCS, vol. 10395, pp. 432–453. Springer (2017)

[7] Beeson, M.: Lambda logic. In: Basin, D.A., Rusinowitch, M. (eds.) IJCAR 2004. LNCS, vol. 3097, pp. 460–474. Springer (2004)

[8] Bentkamp, A., Blanchette, J.C., Cruanes, S., Waldmann, U.: Superposition for lambda-free higher-order logic. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS, vol. 10900, pp. 28–46. Springer (2018)

[9] Benzmüller, C., Kohlhase, M.: System description: LEO—a higher-order theorem prover. In: Kirchner, C., Kirchner, H. (eds.) CADE-15. LNCS, vol. 1421, pp. 139–144. Springer (1998)

[10] Benzmüller, C., Miller, D.: Automation of higher-order logic. In: Siekmann, J.H. (ed.) Computational Logic, Handbook of the History of Logic, vol. 9, pp. 215–254. Elsevier (2014)

[11] Benzmüller, C., Sultana, N., Paulson, L.C., Theiss, F.: The higher-order prover Leo-II. J. Autom. Reason. 55(4), 389–404 (2015)

[12] Bhayat, A., Reger, G.: Set of support for higher-order reasoning. In: Konev, B., Urban, J., Rümmer, P. (eds.) PAAR-2018. CEUR Workshop Proceedings, vol. 2162, pp. 2–16. CEUR-WS.org (2018)

[13] Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. Log. Meth. Comput. Sci. 12(4) (2016)

[14] Blanchette, J.C., Waldmann, U., Wand, D.: A lambda-free higher-order recursive path order. In: Esparza, J., Murawski, A.S. (eds.) FoSSaCS 2017. LNCS, vol. 10203, pp. 461–479. Springer (2017)

[15] Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer (2010)

[16] Brown, C.E.: Satallax: An automatic higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 111–117. Springer (2012)

[17] Cruanes, S.: Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond. PhD thesis, École polytechnique (2015), `https://who.rocq.inria.fr/Simon.Cruanes/files/thesis.pdf`

[18] Cruanes, S.: Superposition with structural induction. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS, vol. 10483, pp. 172–188. Springer (2017)

[19] łLukasz Czajka, Kaliszyk, C.: Hammer for Coq: Automation for dependent type theory (2018)

[20] Czajka, Ł.: Improving automation in interactive theorem provers by efficient encoding of lambda-abstractions. In: Avigad, J., Chlipala, A. (eds.) CPP 2016. pp. 49–57. ACM (2016)

[21] Eggers, A., Kruglov, E., Kupferschmid, S., Scheibler, K., Teige, T., Weidenbach, C.: Superposition modulo non-linear arithmetic. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 119–134. Springer (2011)

[22] Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer (2013)

[23] Huet, G.P.: A mechanization of type theory. In: Nilsson, N.J. (ed.) IJCAI-73. pp. 139–146. William Kaufmann (1973)

[24] Kaliszyk, C., Urban, J.: HOL(y)Hammer: Online ATP service for HOL Light. Math. Comput. Sci. 9(1), 5–22 (2015)

[25] Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer (2007)

[26] Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer (2013)

[27] Lindblad, F.: A focused sequent calculus for higher-order logic. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 61–75. Springer (2014)

[28] Löchner, B.: Things to know when implementing KBO. J. Autom. Reason. 36(4), 289–310 (2006)

[29] Löchner, B., Schulz, S.: An evaluation of shared rewriting. In: de Nivelle, H., Schulz, S. (eds.) IWIL-2001. pp. 33–48. Max-Planck-Institut für Informatik (2001)

[30] McCune, W.: Otter 2.0. In: Stickel, M.E. (ed.) CADE-10. LNCS, vol. 449, pp. 663–664. Springer (1990)

[31] McCune, W.: Experiments with discrimination-tree indexing and path indexing for term retrieval. J. Autom. Reason. 9(2), 147–167 (1992)

[32] Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Autom. Reason. 40(1), 35–60 (2008)

[33] Miller, D.A.: A compact representation of proofs. Studia Logica 46(4), 347–370 (1987)

[34] Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Schulz, S., Ternovska, E. (eds.) IWIL-2010. EPiC, vol. 2, pp. 1–11. EasyChair (2012)

[35] Peltier, N.: A variant of the superposition calculus. Archive of Formal Proofs (2016), `https://www.isa-afp.org/entries/SuperCalc.shtml`

[36] Prevosto, V., Waldmann, U.: SPASS+T. In: Sutcliffe, G., Schmidt, R., Schulz, S. (eds.) ESCoR 2006. CEUR Workshop Proceedings, vol. 192, pp. 18–33. CEUR-WS.org (2006)

[37] Robinson, J.: Mechanizing higher order logic. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 151–170. Edinburgh University Press (1969)

[38] Robinson, J.: A note on mechanizing higher order logic. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 5, pp. 121–135. Edinburgh University Press (1970)

[39] Schulz, S.: E—a brainiac theorem prover. AI Commun. 15(2-3), 111–126 (2002)

[40] Schulz, S.: Fingerprint indexing for paramodulation and rewriting. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 477–483. Springer (2012)

[41] Schulz, S.: Simple and efficient clause subsumption with feature vector indexing. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics—Essays in Memory of William W. McCune. LNCS, vol. 7788, pp. 45–67. Springer (2013)

[42] Schulz, S.: System description: E 1.8. In: McMillan, K.L., Middeldorp, A., Voronkov, A. (eds.) LPAR-19. LNCS, vol. 8312, pp. 735–743. Springer (2013)

[43] Schulz, S.: We know (nearly) nothing!l but can we learn? In: Reger, G., Traytel, D. (eds.) ARCADE 2017, 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements, Gothenburg, Sweden, 6th August 2017. EPiC Series in Computing, vol. 51, pp. 29–32. EasyChair (2017)

[44] Steen, A., Benzmüller, C.: The higher-order prover Leo-III. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS, vol. 10900, pp. 108–116. Springer (2018)

[45] Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 367–373. Springer (2014)

[46] Sultana, N., Blanchette, J.C., Paulson, L.C.: LEO-II and Satallax on the Sledgehammer test bench. J. Applied Logic 11(1), 91–102 (2013)

[47] Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. J. Autom. Reason. 59(4), 483–502 (2017)

[48] Sutcliffe, G.: The 6th IJCAR automated theorem proving system competition—CASC-J6. AI Comm. 26(2), 211–223 (2013)

[49] Sutcliffe, G.: The CADE-26 automated theorem proving system competition—CASC-26. AI Commun. 30(6), 419–432 (2017)

[50] Urban, J., Rudnicki, P., Sutcliffe, G.: ATP and presentation service for Mizar formalizations. J. Autom. Reason. 50(2), 229–241 (2013)

[51] Vukmirović, P.: Implementation of Lambda-Free Higher-Order Superposition. MSc thesis, Vrije Universiteit Amsterdam (2018), `http://matryoshka.gforge.inria.fr/pubs/vukmirovic_msc_thesis.pdf`

[52] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 140–145. Springer (2009)