Extending a brainiac prover to lambda-free higher-order logic

Petar Vukmirović, Jasmin Blanchette, Simon Cruanes, Stephan Schulz

Noname manuscript No.
(will be inserted by the editor)

Extending a Brainiac Prover to Lambda-Free Higher-Order Logic

Petar Vukmirović · Jasmin Blanchette · Simon Cruanes · Stephan Schulz

Received: date / Accepted: date

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 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 toward full higher-order logic.

Keywords Automatic theorem provers · Higher-order logic · First-order logic

1 Introduction

Superposition provers such as E [45], SPASS [57], and Vampire [27] are among the most successful first-order reasoning systems. They serve as backends in various frameworks, including software verifiers (e.g., Why3 [23]), automatic higher-order theorem provers (e.g., Leo-III [46], Satallax [18]), and one-click "hammers" in proof assistants (e.g., HOLyHammer in HOL Light [25], Sledgehammer in Isabelle [36]). Decades of research have gone into refining calculi, devising efficient data structures and algorithms, and developing heuristics to guide proof search [44]. This work has mostly focused on first-order logic with equality.

Research on higher-order automatic provers has resulted in systems such as LEO [11], LEO-II [13], and Leo-III [46], based on resolution and paramodulation, and Satallax [18], based on tableaux and SAT solving. They 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 largely first-order problems, such as those that often arise in interactive verification [48]. For example, at the 2017 installment of the CADE ATP System Competition (CASC) [50], Leo-III, which uses 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 toward full higher-order logic, we initially restrict our focus to a higher-order logic without

Petar Vukmirović
Vrije Universiteit Amsterdam, Amsterdam, the Netherlands
E-mail: p.vukmirovic@vu.nl

Jasmin Blanchette
Vrije Universiteit Amsterdam, Amsterdam, the Netherlands
Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany
Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
E-mail: j.c.blanchette@vu.nl

Simon Cruanes
Aesthetic Integration, Austin, Texas, USA
E-mail: simon@aestheticintegration.com

Stephan Schulz
DHBW Stuttgart, Stuttgart, Germany
E-mail: schulz@eprover.org

λ-expressions (Sect. 2). Compared with first-order logic, its distinguishing features are partial application and applied variables. It is rich enough to express the recursive equations of higher-order combinators, such as map on lists:

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

Our vehicle is E [41, 45], a prover developed primarily by Schulz. It is written in C and offers good performance, with more emphasis on "brainiac" heuristics than on raw speed. E regularly scores among the top systems at CASC and is usually the strongest open-source 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 a prerelease version of E 2.5 configured with the option `--enable-ho`.[1]

The main challenges we faced concerned the representation of types and terms (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 central aspect of our work is a set of techniques we call *prefix optimization*. Higher-order terms contain twice as many proper subterms as first-order terms; for example, $\mathsf{f}\ (\mathsf{g}\ \mathsf{a})\ \mathsf{b}$ contains not only the "argument" subterms $\mathsf{g}\ \mathsf{a}$, a, b but also the "prefix" subterms f, $\mathsf{f}\ (\mathsf{g}\ \mathsf{a})$, g. Many operations, including superposition and rewriting, require traversing all subterms of a term. Using the optimization, the prover traverses subterms recursively in a first-order fashion, considering all the prefixes of a given subterm together. Our experiments (Sect. 9) show that Ehoh is almost as fast as E on first-order problems and can also prove higher-order problems that do not require synthesizing λ-terms. As next steps, we plan to add support for λ-terms and higher-order unification.

An earlier version of this article was presented at TACAS 2019 [55]. This article extends the conference paper with detailed explanations, pseudocode, and correctness proofs. We have also extended E's preprocessor to eliminate Boolean subterms, updated the empirical evaluation, and broadened the treatment of related work.

2 Logic

Our logic is a variant of the intensional λ-free Boolean-free higher-order logic (λfHOL) described by Bentkamp et al. [10, Sect. 2], which could also be called "applicative first-order logic." In the spirit of FOOL [26], we extend the syntax of this logic by erasing the distinction between terms and formulas, and its semantics by interpreting the Boolean type o as a domain of cardinality 2. Functional extensionality can be obtained by adding suitable axioms [10, Sect. 3.1].

[1] `https://github.com/eprover/eprover/commit/80946ac`

A type is either an atomic type ι or a function type $\tau \to \upsilon$, where τ and υ are 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), binary applications $s\ t$, or Boolean terms $\top$, $\bot$, $\neg s$, $s \wedge t$, $s \vee t$, $s \to t$, $s \leftrightarrow t$, $\forall x.\, s$, $\exists x.\, s$, $s \approx t$. Boolean terms are also called *formulas*, and function symbols returning a Boolean value are also called *predicate symbols*. The typing rules are as for the simply typed λ-calculus. A term's *arity* is the number of extra arguments it can take. 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. Subterms are defined in the usual way; for example, $s\ t$ has all subterms of s and t as subterms, in addition to $s\ t$ itself.

Non-Boolean terms have a unique flattened decomposition of the form $\zeta\ s_1 \ldots s_m$, where ζ, the *head*, is a variable or symbol, 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 $\zeta\ s_1 \ldots s_m$. 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 have the same type, or its negation $s \not\approx t$. Clauses C, D are finite multisets of literals, written $L_1 \vee \cdots \vee L_n$. E and Ehoh clausify the input as a preprocessing step, producing a clause set in which the only proper Boolean subterms are variables, $\top$, and $\bot$.

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 s_i has the same type as x_i. The substitution $\sigma[x \mapsto s]$ maps x to s and otherwise coincides with σ. Applying σ to a variable beyond σ's domain is the identity. Composition $(\sigma' \circ \sigma)(t)$ is defined as $\sigma'(\sigma(t))$.

A well-known technique to support λfHOL is to use the *applicative encoding*: Every n-ary symbol is mapped to a nullary symbol, and application is represented by a distinguished binary symbol @. Thus, 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, since it also introduces @'s for terms within λfHOL's first-order fragment. By doubling the size and depth of terms, the encoding clutters data structures and slows down term traversals. In our empirical evaluation, we find that the applicative encoding can decrease the success rate by up to 15% (Sect. 9). For these and further reasons, it is not ideal (Sect. 10).

3 Types and Terms

The term representation is a central concern when building a theorem prover. Delicate changes to E's 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.

Types For most of its history, E supported only untyped first-order logic. Cruanes implemented support for atomic

types for E 2.0 [19, p. 117]. Symbols are declared with a type signature: $\mathsf{f} : \tau_1 \times \cdots \times \tau_m \to \tau$. Atomic types are represented by integers, leading to efficient type comparisons.

In λfHOL, a type signature is simply a type τ, in which the type constructor $\to$ can be nested—e.g., $(\iota \to \iota) \to \iota$. A natural way to represent such types is to mimic their recursive structure using a tagged union. However, this leads to memory fragmentation; 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 $\to$ and pointing to the array $(\tau_1, \ldots, \tau_n, \iota)$.

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.

Terms In E, terms are stored as perfectly shared directed acyclic graphs [30]. 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; `args`, an array of size `arity` consisting of pointers to arguments; and `binding`, which may store a substitution for a variable (if `f_code` < 0), used for unification and matching.

In first-order logic, the arity of a variable is always 0, and the arity of a symbol 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, for symbols let `arity` be the number of actual arguments, and rename the field `num_args`. This representation, often called "spine notation," is isomorphic to the standard definition of higher-order terms with binary application. It is employed in various higher-order reasoning systems, including Leo-III [46] and Zipperposition [9].

A side effect of the flattened representation is that prefix subterms are not shared. For example, the terms $\mathsf{f}\ \mathsf{a}$ and $\mathsf{f}\ \mathsf{a}\ \mathsf{b}$ 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 $\mathsf{f}\ \mathsf{a}$. 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, $x\ \mathsf{b}\ \mathsf{c}$ is represented by the cell $@(x,\mathsf{b},\mathsf{c})$, where x is a shared subcell. This is graceful, since variables never occur applied in first-order terms. The main drawback is that some normalization is necessary after substitution: Whenever a variable is instantiated by a symbol-headed term, the @ symbol must be eliminated. Applying the substitution $\{x \mapsto \mathsf{f}\ \mathsf{a}\}$ to the cell $@(x,\mathsf{b},\mathsf{c})$ must produce $\mathsf{f}(\mathsf{a},\mathsf{b},\mathsf{c})$ and not $@(\mathsf{f}(\mathsf{a}),\mathsf{b},\mathsf{c})$, for consistency with other occurrences of $\mathsf{f}\ \mathsf{a}\ \mathsf{b}\ \mathsf{c}$.

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 x's `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.

Term Orders Superposition provers rely on term orders to prune the search space. The order must be a simplification order that is total on variable-free terms. E implements both the Knuth–Bendix order (KBO) and the lexicographic path order (LPO). KBO is widely regarded as the more robust option for superposition. In earlier work, Blanchette and colleagues have shown that only KBO can be generalized gracefully while preserving the necessary properties for superposition [7, 16]. For this reason, we focus on KBO.

E implements Löchner's linear-time algorithm for KBO [29], which relies on the tupling method to store intermediate results. It is straightforward to generalize the algorithm to compute the graceful λfHOL version of KBO [7]. 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 $\overline{t}_n$.

Input and Output Syntax E implements the TPTP [51] formats FOF and TF0, corresponding to untyped and monomorphic first-order logic, for both input and output. In Ehoh, we added support for the λfHOL fragment of TPTP TH0, which provides monomorphic higher-order logic. Thanks to the use of a standard format, Ehoh's proofs can immediately be parsed by Sledgehammer [36], which reconstructs them using a variety of techniques. There is ongoing work on increasing the level of detail of E's proofs, to facilitate proof interchange and independent proof checking [38]; this will also benefit Ehoh.

4 Unification and Matching

Syntactic unification of (Boolean-free) λfHOL terms has a first-order flavor. It is decidable, and most general unifiers (MGUs) are unique up to variable renaming. For example, the unification constraint $\mathsf{f}\ (y\ \mathsf{a}) \stackrel{?}{=} y\ (\mathsf{f}\ \mathsf{a})$ has the MGU $\{y \mapsto \mathsf{f}\}$, whereas in full higher-order logic infinitely many independent solutions of the form $\{y \mapsto \lambda x.\ \mathsf{f}\ (\mathsf{f}\ (\cdots(\mathsf{f}\ x)\cdots))\}$ exist. 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. 2), perform first-order unification or matching, and decode the resulting substitution. To avoid the overhead, we generalize the first-order unification and matching procedures to operate directly on λfHOL terms.

Unification We present our unification procedure as a nondeterministic transition system that generalizes Baader and Nipkow [5]. A unification problem consists of a finite set S of unification constraints $s_i \stackrel{?}{=} t_i$, where s_i and t_i are of the same type. A problem is in *solved form* if it has the form $\{x_1 \stackrel{?}{=} t_1, \ldots, x_n \stackrel{?}{=} t_n\}$, 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 $\quad \{t \stackrel{?}{=} t\} \uplus S \Longrightarrow S$

Decompose $\quad \{\mathsf{f}\ \overline{s}_m \stackrel{?}{=} \mathsf{f}\ \overline{t}_m\} \uplus S \Longrightarrow S \cup \{s_1 \stackrel{?}{=} t_1, \ldots, s_m \stackrel{?}{=} t_m\}$

DecomposeX $\quad \{x\ \overline{s}_m \stackrel{?}{=} u\ \overline{t}_m\} \uplus S \Longrightarrow S \cup \{x \stackrel{?}{=} u,\ s_1 \stackrel{?}{=} t_1, \ldots, s_m \stackrel{?}{=} t_m\}$ if x and u have the same type and $m > 0$

Orient $\quad \{\mathsf{f}\ \overline{s} \stackrel{?}{=} x\ \overline{t}\} \uplus S \Longrightarrow S \cup \{x\ \overline{t} \stackrel{?}{=} \mathsf{f}\ \overline{s}\}$

OrientXY $\quad \{x\ \overline{s}_m \stackrel{?}{=} y\ \overline{t}_n\} \uplus S \Longrightarrow S \cup \{y\ \overline{t}_n \stackrel{?}{=} x\ \overline{s}_m\}$ if $m > n$

Eliminate $\quad \{x \stackrel{?}{=} t\} \uplus S \Longrightarrow \{x \stackrel{?}{=} t\} \cup \{x \mapsto t\}(S)$ 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 $\quad \{\mathsf{f}\ \overline{s} \stackrel{?}{=} \mathsf{g}\ \overline{t}\} \uplus S \Longrightarrow \bot$ if $\mathsf{f} \neq \mathsf{g}$

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

ClashLenXF $\quad \{x\ \overline{s}_m \stackrel{?}{=} \mathsf{f}\ \overline{t}_n\} \uplus S \Longrightarrow \bot$ if $m > n$

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

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

The derivation below demonstrates the computation of MGUs for the unification problem $\{x\ (z\ \mathsf{b}\ \mathsf{c}) \stackrel{?}{=} \mathsf{g}\ \mathsf{a}\ (y\ \mathsf{c})\}$:

$$
\begin{aligned}
& \{x\ (z\ \mathsf{b}\ \mathsf{c}) \stackrel{?}{=} \mathsf{g}\ \mathsf{a}\ (y\ \mathsf{c})\} \\
\Longrightarrow_{\mathsf{DecomposeX}}\ & \{x \stackrel{?}{=} \mathsf{g}\ \mathsf{a},\ z\ \mathsf{b}\ \mathsf{c} \stackrel{?}{=} y\ \mathsf{c}\} \\
\Longrightarrow_{\mathsf{OrientXY}}\ & \{x \stackrel{?}{=} \mathsf{g}\ \mathsf{a},\ y\ \mathsf{c} \stackrel{?}{=} z\ \mathsf{b}\ \mathsf{c}\} \\
\Longrightarrow_{\mathsf{DecomposeX}}\ & \{x \stackrel{?}{=} \mathsf{g}\ \mathsf{a},\ y \stackrel{?}{=} z\ \mathsf{b},\ \mathsf{c} \stackrel{?}{=} \mathsf{c}\} \\
\Longrightarrow_{\mathsf{Delete}}\ & \{x \stackrel{?}{=} \mathsf{g}\ \mathsf{a},\ y \stackrel{?}{=} z\ \mathsf{b}\}
\end{aligned}
$$

E 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 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 virtually 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. 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 $\mathcal{U}(S \cup S') = \mathcal{U}(S) \cap \mathcal{U}(S')$. 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, Decompose, Orient, and Eliminate are proved as in Baader and Nipkow. OrientXY trivially preserves unifiers. For DecomposeX, the core of the argument is as follows:

$$
\begin{aligned}
& \sigma \in \mathcal{U}(\{x\ \overline{s}_m \stackrel{?}{=} u\ \overline{t}_m\}) \\
\text{iff}\quad & \sigma(x\ \overline{s}_m) = \sigma(u\ \overline{t}_m) \\
\text{iff}\quad & \sigma(x)\ \sigma(s_1) \ldots \sigma(s_m) = \sigma(u)\ \sigma(t_1) \ldots \sigma(t_m) \\
\text{iff}\quad & \sigma(x) = \sigma(u),\ \sigma(s_1) = \sigma(t_1),\ \ldots,\ \text{and}\ \sigma(s_m) = \sigma(t_m) \\
\text{iff}\quad & \sigma \in \mathcal{U}(\{x \stackrel{?}{=} u,\ s_1 \stackrel{?}{=} t_1, \ldots, s_m \stackrel{?}{=} t_m\})
\end{aligned}
$$

The proof of the problem's unsolvability if rule Clash or OccursCheck is applicable carries over from Baader and Nipkow. For ClashTypeX, the justification 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 have the same type. Similarly, for 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 $s \stackrel{?}{=} t \in S$. 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 $x \in \mathcal{V}ar(t)$: OccursCheck is applicable.
- SUBCASE $t \neq x$, $x \notin \mathcal{V}ar(t)$, and $x \in \mathcal{V}ar(S \setminus \{s \stackrel{?}{=} t\})$: Eliminate is applicable.
- SUBCASE $t \neq x$, $x \notin \mathcal{V}ar(t)$, and $x \notin \mathcal{V}ar(S \setminus \{s \stackrel{?}{=} t\})$: The variable x is solved in S.

CASE $s = x\ \overline{s}_m$ for $m > 0$:

- SUBCASE $t = \eta\ \overline{t}_n$ for $n \geq m$: DecomposeX or ClashTypeX is applicable, depending on whether x and $\eta\ \overline{t}_{n-m}$ have the same type.
- SUBCASE $t = y\ \overline{t}_n$ for $n < m$: OrientXY is applicable.
- SUBCASE $t = \mathsf{f}\ \overline{t}_n$ for $n < m$: 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, $m = n$. Decompose is applicable.
- SUBCASE $t = \mathsf{g}\ \overline{t}_n$: Clash is applicable.

Since each constraint is of the form $x \stackrel{?}{=} t$ 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. The second part follows from Lemma 1 and Lemmas 2 and 3. □

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

Proof We define an auxiliary notion of weight: $\mathcal{W}(\zeta\ \overline{s}_m) = m + 1 + \sum_{i=1}^{m} \mathcal{W}(s_i)$. 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, $\sum_{s \stackrel{?}{=} t \in S} \mathcal{W}(s) + \mathcal{W}(t)$; n_3 is the number of right-hand sides with variable heads, $|\{s \stackrel{?}{=} x\ \overline{t} \in S\}|$; and n_4 is the number of arguments to left-hand side variable heads, $\sum_{x\ \overline{s}_m \stackrel{?}{=} t \in S} m$.

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	$>$			

The negative rules, which produce the special value $\bot$, cannot contribute to an infinite $\Longrightarrow$ chain. □

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 its *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()
      ∧ (¬ s.head.isVar()
        ∨ s.num_args > t.num_args)

function DEREF(Term s) is
  while s.head.isVar() ∧ s.head.binding ≠ Null do
    s ← APPLYSUBST(s, s.head.binding)
  return s

function GOBBLEPREFIX(Term x, Term t) is
  res ← Null
  if x.type.args is suffix of t.head.type.args then
    pref_len ← t.head.type.arity − x.type.arity
    if pref_len ≤ t.num_args then
      res ← TERM(t.head, t.args[1..pref_len])
  return res

function UNIFY(Term s, Term t) is
  constraints ← DOUBLEENDEDQUEUE()
  constraints.prepend(s)
  constraints.prepend(t)
  while ¬ constraints.isEmpty() do
```

```
t ← DEREF(constraints.dequeue())
s ← DEREF(constraints.dequeue())
if s ≠ t then
  if SWAPNEEDED(s,t) then
    (t,s) ← (s,t)
  if s.head.isVar() then
    x ← s.head
    prefix ← GOBBLEPREFIX(x,t)
    if prefix ≠ Null then
      start_idx ← prefix.num_args + 1
      if x occurs in prefix then
        return False
      else
        x.binding ← prefix
    else
      return False
  else if s.head = t.head then
    start_idx ← 1
  else
    return False
  for i ← start_idx to t.num_args do
    s_arg ← s.args[i − start_idx + 1]
    t_arg ← t.args[i]
    if (s_arg.head.isVar()
        ∨ t_arg.head.isVar()) then
      constraints.append(t_arg)
      constraints.append(s_arg)
    else
      constraints.prepend(s_arg)
      constraints.prepend(t_arg)
return True
```

Matching Given 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." We are interested in most general generalizations (MGGs). Matching can be reduced to unification by treating variables in t as nullary symbols [5], but E implements it separately.

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

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

ClashLenXY $\quad \{x\ \overline{s}_m \lesssim^? y\ \overline{t}_n\} \uplus S \Longrightarrow \bot$
if $x \neq y$ and $m > n$

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

The matching relation is sound, complete, and well founded. Interestingly, a Delete rule would be unsound for matching. Consider the problem $\{x \lesssim^? x, x \lesssim^? \mathsf{g}\ x\}$. Applying Delete to the first constraint would yield the solution $\{x \lesssim^? \mathsf{g}\ x\}$, even though the original problem is clearly unsolvable.

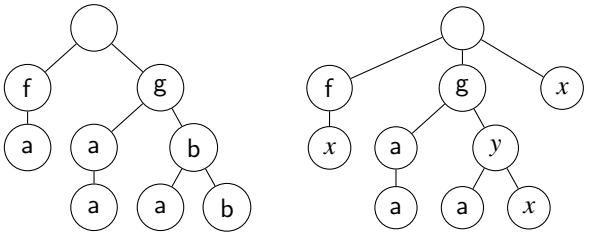
5 Indexing Data Structures

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 [32] and fingerprint indices [42], that needed to be generalized to λfHOL. It also uses feature vector indices [43] to speed up subsumption and related techniques, but these require no changes to work with λfHOL.

Discrimination Trees Discrimination trees [32] 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 D_1 and D_2:



Assuming $\mathsf{a}, \mathsf{b}, x, y : \iota$, $\mathsf{f} : \iota \to \iota$, and $\mathsf{g} : \iota^2 \to \iota$, D_1 represents the term set $\{\mathsf{f}(\mathsf{a}), \mathsf{g}(\mathsf{a},\mathsf{a}), \mathsf{g}(\mathsf{b},\mathsf{a}), \mathsf{g}(\mathsf{b},\mathsf{b})\}$, and D_2 represents the term set $\{\mathsf{f}(x), \mathsf{g}(\mathsf{a},\mathsf{a}), \mathsf{g}(y,\mathsf{a}), \mathsf{g}(y,x), x\}$. E uses perfect discrimination trees for finding generalizations of query terms. Thus, if the query term is $\mathsf{g}(\mathsf{a},\mathsf{a})$, it would follow the path g.a.a in 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 $\{\mathsf{g}(\mathsf{a},\mathsf{a}), \mathsf{g}(y,\mathsf{a}), \mathsf{g}(y,x), x\}$.

It is crucial for this data structure that distinct terms always give rise to distinct serialized terms. Conveniently, this property also holds for λfHOL terms. Suppose 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.

When generalizing the data structure to λfHOL, we face a 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 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. This is achieved using two stacks that store subterms of the query term: T stores the terms that must be matched in turn against the current subtree, and P stores, for each node from the root to the current subtree, the corresponding processed term.

Let $[a_1,\ldots,a_n]$ denote an n-item stack with a_1 on top. Given a query term t, the matching procedure starts at the root with $\sigma = \emptyset$, $T = [t]$, and $P = []$. The procedure advances by repeatedly moving to a suitable child node:

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

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 the term t from T and push it onto P.

The goal is to reach an accepting node. If the query term and all the terms stored in the tree are first-order, T will then be empty, and the entire query term will have been matched. Backtracking works in reverse: Pop a term t from P; if the current node is labeled with an n-ary symbol, discard T's topmost n items; push t onto T. Undo any variable bindings.

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 g.b.a:

	ϵ	g	g.b	g.b.a
σ:	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
T:	[g(b, a)]	[b, a]	[a]	[]
P:	[]	[g(b, a)]	[b, g(b, a)]	[a, b, g(b, a)]

Backtracking amounts to moving leftward: To get back from g to the root, we pop $\mathsf{g}(\mathsf{b},\mathsf{a})$ from P, we discard two items from T, and we push $\mathsf{g}(\mathsf{b},\mathsf{a})$ onto T.

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 head ζ matches a prefix t' of the k-ary term t popped from T and leaves $n-k$ arguments $\bar{u}$ to be pushed back, with $t = t'\ \bar{u}$. If ζ is a variable, it must be bound to the prefix t' assuming ζ and t' are of same type. Backtracking works analogously: Given the arity n of the node label ζ and the arity k of the term t popped from P, we discard the topmost $n-k$ items $\bar{u}$ from P.

To illustrate the procedure, we consider the tree D_2 but change y's type to $\iota \to \iota$. This tree stores $\{\mathsf{f}\ x, \mathsf{g}\ \mathsf{a}\ \mathsf{a}, \mathsf{g}\ (y\ \mathsf{a}), \mathsf{g}\ (y\ x), x\}$. Let $\mathsf{g}\ (\mathsf{g}\ \mathsf{a}\ \mathsf{b})$ be the query term. We have the following sequence of substitutions σ and stacks T, P:

ϵ	g	g.y	g.y.x
$\emptyset$	$\emptyset$	$\{y \mapsto \mathsf{g}\ \mathsf{a}\}$	$\{y \mapsto \mathsf{g}\ \mathsf{a}, x \mapsto \mathsf{b}\}$
[g (g a b)]	[g a b]	[b]	[]
[]	[g (g a b)]	[g a b, g (g a b)]	[b, g a b, g (g a b)]

When backtracking from g.y to g, by comparing y's arity of $n = 1$ with g a b's arity of $k = 0$, we determine that one item must be discarded from T. Finally, to avoid traversing twice as many subterms as in the first-order case, we can optimize prefixes: Given a query term $\zeta\ \bar{t}_n$, we can also match prefixes $\zeta\ \bar{t}_k$, where $k < n$, by allowing T to be nonempty when we reach an accepting node.

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

Let D be a perfect discrimination tree. $\mathcal{T}erm(D)$ denotes the set of terms stored in D. The function $D|_\zeta$ returns the child of D labeled with ζ, if it exists. Child nodes are themselves perfect discrimination (sub)trees. Given any node D, if the node is accepting, then the value stored on that node is defined as $val(D) = (s, d)$, where s is the accepted term and d is some arbitrary data; otherwise, $val(D)$ is undefined.

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 $(\mathsf{f}\ \bar{s}_m \cdot \bar{t}, \bar{b}, D, \sigma) \rightsquigarrow (\bar{s}_m \cdot \bar{t}, (\mathsf{f}\ \bar{s}_m, D, \sigma) \cdot \bar{b}, D|_\mathsf{f}, \sigma)$
if $D|_\mathsf{f}$ is defined

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

Backtrack $(\bar{s}_m \cdot \bar{t}, (s, D_0, \sigma_0) \cdot \bar{b}, D, \sigma) \rightsquigarrow (s \cdot \bar{t}, \bar{b}, D_0, \sigma_0)$
if $D_0|_\zeta = D$ and $m = arity(\zeta) - arity(s)$

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

Above, $\cdot$ denotes prepending an element or a list to a list. Intuitively, AdvanceF and AdvanceX move deeper in the tree, generalizing cases A and B above to λfHOL terms. Backtrack can be used to return to a previous state. Success extracts the term t and data d stored in an accepting node.

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

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

Let $\rightsquigarrow_{\mathsf{Advance}} = \rightsquigarrow_{\mathsf{AdvanceF}} \cup \rightsquigarrow_{\mathsf{AdvanceX}}$. It is easy to show that Backtrack undoes an Advance transition:

Lemma 6 *If $\mathcal{Q} \rightsquigarrow_{\mathsf{Advance}} \mathcal{Q}'$, then $\mathcal{Q}' \rightsquigarrow_{\mathsf{Backtrack}} \mathcal{Q}$.*

Proof For both Advance steps, we show that Backtrack restores the state properly. If AdvanceF was applied, we have

$$
\begin{aligned}
(\mathsf{f}\ \overline{s}_m \cdot \overline{t}, \overline{b}, D, \sigma) & \rightsquigarrow_{\mathsf{AdvanceF}} (\overline{s}_m \cdot \overline{t}, (\mathsf{f}\ \overline{s}_m, D, \sigma) \cdot \overline{b}, D|_{\mathsf{f}}, \sigma) \\
& \rightsquigarrow_{\mathsf{Backtrack}} (\overline{t'}, \overline{b}, D, \sigma)
\end{aligned}
$$

We must show that $\overline{t'} = \mathsf{f}\ \overline{s}_m \cdot \overline{t}$. Let $k = \mathit{arity}(\mathsf{f})$ and $l = \mathit{arity}(\mathsf{f}\ \overline{s}_m)$. By definition of k, we have $m = k - l$, as in Backtrack's side condition. Thus, $\overline{t'} = \mathsf{f}\ \overline{s}_m \cdot \overline{t}$. The other case is

$$
\begin{aligned}
(s\ \overline{s}_m \cdot \overline{t}, \overline{b}, D, \sigma) & \rightsquigarrow_{\mathsf{AdvanceX}} (\overline{s}_m \cdot \overline{t}, (s\ \overline{s}_m, D, \sigma) \cdot \overline{b}, D|_x, \sigma') \\
& \rightsquigarrow_{\mathsf{Backtrack}} (\overline{t'}, \overline{b}, D, \sigma)
\end{aligned}
$$

where $\sigma' = \sigma[x \mapsto s]$. Again, we must show that $\overline{t'} = s\ \overline{s}_m \cdot \overline{t}$. Terms x and s must have the same type for AdvanceX to be applicable; therefore, they have the same arity. Then, we conclude $m = \mathit{arity}(s) - \mathit{arity}(s\ \overline{s}_m) = \mathit{arity}(x) - \mathit{arity}(s\ \overline{s}_m)$, as in Backtrack's side condition. Thus, $\overline{t'} = s\ \overline{s}_m \cdot \overline{t}$. □

Lemma 7 *If $\mathcal{Q} \rightsquigarrow_{\mathsf{Advance}} \mathcal{Q}' \rightsquigarrow_{\mathsf{Backtrack}} \mathcal{Q}''$, then $\mathcal{Q}'' = \mathcal{Q}$.*

Proof By Lemma 6, $\mathcal{Q}' \rightsquigarrow_{\mathsf{Backtrack}} \mathcal{Q}$. Furthermore, Backtrack is clearly functional. Thus, $\mathcal{Q}'' = \mathcal{Q}$. □

Lemma 8 *Let $\mathcal{Q} = ([t], [], D, \emptyset)$. If $\mathcal{Q} \rightsquigarrow^* \mathcal{Q}'$, then $\mathcal{Q} \rightsquigarrow^*_{\mathsf{Advance}} \mathcal{Q}'$.*

Proof Let $\mathcal{Q} = \mathcal{Q}_0 \rightsquigarrow \cdots \rightsquigarrow \mathcal{Q}_n = \mathcal{Q}'$. Let i be the index of the first transition of the form $\mathcal{Q}_i \rightsquigarrow_{\mathsf{Backtrack}} \mathcal{Q}_{i+1}$. 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 7, $\mathcal{Q}_{i-1} = \mathcal{Q}_{i+1}$. 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 9 *There exist no infinite chains of the form $\mathcal{Q}_0 \rightsquigarrow_{\mathsf{Advance}} \mathcal{Q}_1 \rightsquigarrow_{\mathsf{Advance}} \cdots$.*

Proof With each Advance transition, the height of the discrimination tree decreases by at least one. □

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

$$
\begin{aligned}
&\mathsf{Decompose} && (\mathsf{f}\ \overline{s}_m \lesssim^? \mathsf{f}\ \overline{t}_m \cdot \overline{c}, \sigma) \hookrightarrow ((s_1 \lesssim^? t_1, \ldots, s_m \lesssim^? t_m) \cdot \overline{c}, \sigma) \\
&\mathsf{DecomposeX} && (x\ \overline{s}_m \lesssim^? u\ \overline{t}_m \cdot \overline{c}, \sigma) \hookrightarrow ((s_1 \lesssim^? t_1, \ldots, s_m \lesssim^? t_m) \cdot \overline{c}, \sigma[x \mapsto u]) \\
& && \quad \text{if } x \text{ and } u \text{ have the same type and either } \sigma(x) \text{ is undefined or } \sigma(x) = u \\
&\mathsf{Success} && ([], \sigma) \hookrightarrow \sigma \\
&\mathsf{Clash} && (\mathsf{f}\ \overline{s}_m \lesssim^? \mathsf{g}\ \overline{t}_n \cdot \overline{c}, \sigma) \hookrightarrow \bot \\
&\mathsf{ClashTypeX} && (x\ \overline{s}_m \lesssim^? u\ \overline{t}_m \cdot \overline{c}, \sigma) \hookrightarrow \bot \\
& && \quad \text{if } x \text{ and } u \text{ have different types} \\
&\mathsf{ClashLenXF} && (x\ \overline{s}_m \lesssim^? \mathsf{f}\ \overline{t}_n \cdot \overline{c}, \sigma) \hookrightarrow \bot \quad \text{if } m > n \\
&\mathsf{ClashLenXY} && (x\ \overline{s}_m \lesssim^? y\ \overline{t}_n \cdot \overline{c}, \sigma) \hookrightarrow \bot \\
& && \quad \text{if } x \neq y \text{ and } m > n \\
&\mathsf{ClashFX} && (\mathsf{f}\ \overline{s} \lesssim^? x\ \overline{t} \cdot \overline{c}, \sigma) \hookrightarrow \bot \\
&\mathsf{Double} && (x\ \overline{s}_m \lesssim^? u\ \overline{t}_m \cdot \overline{c}, \sigma) \hookrightarrow \bot \\
& && \quad \text{if } x \text{ and } u \text{ have the same type, } \sigma(x) \text{ is defined, and } \sigma(x) \neq u
\end{aligned}
$$

We need an auxiliary function to convert $\hookrightarrow$ states to $\Longrightarrow$ states. Let $\alpha(\{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}) = \{x_1 \lesssim^? s_1, \ldots, x_m \lesssim^? s_m\}$, $\alpha(\overline{c}, \sigma) = \{c \mid c \in \overline{c}\} \cup \alpha(\sigma)$, and $\alpha(\bot) = \bot$. 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 10 *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 $\mathcal{R} = (\overline{c'}, \sigma')$: Only $\hookrightarrow_{\mathsf{Decompose}}$ and $\hookrightarrow_{\mathsf{DecomposeX}}$ are possible. If $\hookrightarrow_{\mathsf{Decompose}}$ is applied, then $\Longrightarrow_{\mathsf{Decompose}}$ is applicable and results in $\alpha(\mathcal{R})$. If $\hookrightarrow_{\mathsf{DecomposeX}}$ is applied, we have either $m > 0$, 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. $\hookrightarrow_{\mathsf{Double}}$ corresponds to $\Longrightarrow_{\mathsf{Double}}$ if $m = 0$. If $m \neq 0$, we need an intermediate $\Longrightarrow_{\mathsf{DecomposeX}}$ step before $\Longrightarrow_{\mathsf{Double}}$ can be applied to derive $\bot$. Since $\hookrightarrow_{\mathsf{Double}}$ is applicable, $\sigma(x) = u' \neq u$. Hence, $x \lesssim^? u'$ must be present in $\alpha(\overline{c}, \sigma)$. $\Longrightarrow_{\mathsf{DecomposeX}}$ 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 11 *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. It necessarily focuses on c_1, since the constraints from $\alpha(\sigma')$ correspond to solved variables. In all cases except for $\Longrightarrow_{\mathsf{DecomposeX}}$, a homologous $\hookrightarrow$ rule can be applied to $\mathcal{S}'$. If $\Longrightarrow_{\mathsf{DecomposeX}}$ would make $\Longrightarrow_{\mathsf{Double}}$ applicable, then we can apply $\hookrightarrow_{\mathsf{Double}}$ to $\mathcal{S}'$; otherwise, $\hookrightarrow_{\mathsf{DecomposeX}}$ is applicable.

Second, by Lemma 10, if $\mathcal{S} \hookrightarrow^! \sigma'$, then $\alpha(\mathcal{S}) \Longrightarrow^* \alpha(\sigma')$. By construction, $\alpha(\sigma')$ is in solved form. Therefore, $\alpha(\mathcal{S}) \Longrightarrow^! \alpha(\sigma')$. By completeness of $\Longrightarrow$, the substitution corresponding to $\alpha(\sigma')$—that is, σ'—is the MGG of $\alpha(\mathcal{S})$.

Third, by Lemma 10, if $\mathcal{S} \hookrightarrow^! \bot$, then $\alpha(\mathcal{S}) \Longrightarrow^! \bot$. By soundness of $\Longrightarrow$, $\alpha(\mathcal{S})$ has no solutions. □

Lemma 12 *The relation $\hookrightarrow$ is well founded.*

Proof By Lemma 10, every $\hookrightarrow$ transition corresponds to zero or more $\Longrightarrow$ transitions. Since $\Longrightarrow$ is well founded, the only transitions that can violate well-foundedness of $\hookrightarrow$ are the ones that take idle $\Longrightarrow^*$ transitions: $\hookrightarrow_{\mathsf{DecomposeX}}$ for $m = 0$ and $\hookrightarrow_{\mathsf{Success}}$. The latter is terminal, so it cannot contribute to infinite chains. As for $\hookrightarrow_{\mathsf{DecomposeX}}$, with $m = 0$, it decreases the following measure μ, which the other rules nonstrictly decrease, with respect to the multiset extension of $<$ on natural numbers: $\mu([s_1 \lesssim^? t_1, \ldots, s_m \lesssim^? t_m], \sigma) = \{|s_1|, \ldots |s_m|\}$, where $|s|$ denotes the syntactic size of s. □

Lemma 13 *If term s generalizes t, then $([s \lesssim^? t], \emptyset) \hookrightarrow^! \sigma$, where σ is the MGG of $s \lesssim^? t$.*

Proof By Lemma 12, there exists a normal form $\mathcal{R}$ starting from $\mathcal{S} = ([s \lesssim^? t], \emptyset)$. Since $s \lesssim^? t$ is solvable, by Lemma 11, and soundness of $\hookrightarrow$ (a consequence of Lemma 10 and soundness of $\Longrightarrow$), $\mathcal{R}$ must be the MGG for s and t. □

Lemma 14 *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 13, 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 $([s \lesssim^? t], \emptyset) = (\overline{c_0}, \sigma_0) \hookrightarrow \cdots \hookrightarrow (\overline{c_n}, \sigma_n) \hookrightarrow \sigma$. The n first transitions must be Decompose or DecomposeX, and the last transition must be Success.

We show that there exists a derivation of the form $([t], [\,], D, \emptyset) = Q_0 \rightsquigarrow \cdots \rightsquigarrow Q_n \rightsquigarrow ((s,d), \sigma)$, where $Q_i = (\overline{t_i}, \overline{b_i}, D_i, \sigma_i)$ for each i. We define $\overline{t_i}$, $\overline{b_i}$, and D_i as follows, for $i > 0$. The list $\overline{t_i}$ consists of the right-hand sides of the constraints $\overline{c_i}$, in the same order. Let *hd* be the function that extracts the head of a list. We set $b_i = (hd(\overline{t_{i-1}}), D_{i-1}, \sigma_{i-1})$. We know that $\overline{c_{i-1}}$ is nonempty, since there exists a transition $(\overline{c_{i-1}}, \sigma_{i-1}) \hookrightarrow (\overline{c_i}, \sigma_i)$; thus, $\overline{t_{i-1}}$ is nonempty. If an accepting node storing s was reached in n steps, the serialization of s must be of the form $\zeta_1 . \cdots . \zeta_n$. Take $D_i = D_{i-1}|_{\zeta_i}$.

The sequence of states Q_i forms a derivation: If $(\overline{c_i}, \sigma_i) \hookrightarrow_{\mathsf{Decompose}} (\overline{c_{i+1}}, \sigma_{i+1})$, then $Q_i \rightsquigarrow_{\mathsf{AdvanceF}} Q_{i+1}$. If $(\overline{c_i}, \sigma_i) \hookrightarrow_{\mathsf{DecomposeX}} (\overline{c_{i+1}}, \sigma_{i+1})$, then $Q_i \rightsquigarrow_{\mathsf{AdvanceX}} Q_{i+1}$. If $(\overline{c_n}, \sigma_n) \hookrightarrow_{\mathsf{Success}} \sigma$, then $Q_n \rightsquigarrow_{\mathsf{Success}} ((s,d), \sigma)$. □

Lemma 15 *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 $([t], [\,], D, \emptyset) = Q_0 \rightsquigarrow \cdots \rightsquigarrow Q_n \rightsquigarrow ((s,d), \sigma)$ be a derivation, where $Q_i = (\overline{t_i}, \overline{b_i}, D_i, \sigma_i)$ for each i. Without loss of generality, by Lemma 8, we can assume that the derivation contains no Backtrack transitions.

The first conjunct, $s \in \mathcal{T}erm(D)$, clearly holds 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: $preord([\,]) = [\,]$ and $preord(\zeta\ \overline{s_n} \cdot \overline{xs}) = (\zeta,\ \overline{s_n} \cdot \overline{xs}) \cdot preord(\overline{s_n} \cdot \overline{xs})$. Given a term s, $preord([s])$ gives a sequence $(\zeta_1, \overline{u_1}), \ldots, (\zeta_n, \overline{u_n})$. Since $s \in \mathcal{T}erm(D)$, the sequence $D_0, \ldots, D_n$ follows the preorder serialization of s: $D_i = D_{i-1}|_{\zeta_i}$ for $i > 0$.

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 the elements of $\overline{u_i}$ and right-hand sides are the elements of $\overline{t_i}$, in the order they appear in the respective lists. By inspecting the definition of *preord* and the changes each Advance step makes to the head of $\overline{t_i}$, we can see that $\overline{u_i}$ and $\overline{t_i}$ have the same length. The sequence of states $\mathcal{S}_i$ forms a derivation: If $Q_i \rightsquigarrow_{\mathsf{AdvanceF}} Q_{i+1}$, then $\mathcal{S}_i \hookrightarrow_{\mathsf{Decompose}} \mathcal{S}_{i+1}$. If $Q_i \rightsquigarrow_{\mathsf{AdvanceX}} Q_{i+1}$, then $\mathcal{S}_i \hookrightarrow_{\mathsf{DecomposeX}} \mathcal{S}_{i+1}$. If $Q_n \rightsquigarrow_{\mathsf{Success}} \sigma$, then $\mathcal{S}_n \hookrightarrow_{\mathsf{Success}} \sigma$. □

Theorem 16 (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 14 and 15. □

The theorem tells us that given a term t, all generalizations s 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 types of applications, E provides iterators that store the state of a traversal. After an 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 following definitions constitute the high-level interface for iterating through values incrementally or for obtaining all values of nodes that store generalizations of the query term in D.

```
function INITITER(PDTNode D, Term t) is
  i ← ITERATOR()
  (i.node, i.t_stack, i.t_proc, i.c_iter) ← (D, [t], [], Start)
  return i

procedure FINDNEXTVAL(Iterator i) is
  do
    FINDNEXTNODE(i)
  while i.node ≠ Null ∧
      (¬i.t_stack.isEmpty() ∨ ¬i.node.has_val())

function ALLVALS(PDTNode D, Term t) is
  i ← INITITER(D, t)
  FINDNEXTVAL(i)
  res ← ∅
  while i.node ≠ Null do
    res ← res ∪ {i.node.val()}
    FINDNEXTVAL(i)
  return res
```

The core functionality is implemented in FINDNEXTNODE, presented below. This procedure moves the iterator to the next node that has not been explored in the search for generalization, or *Null* if the entire tree has been traversed. It first goes through all child nodes labeled with a variable before possibly visiting the child node labeled with a function symbol. 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 representing the next variable. 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; *End* indicates that we have visited all children. Finally, the expression $n.child(\zeta)$ returns a child of the node n labeled ζ if such a child exists or *Null* otherwise.

```
procedure FINDNEXTNODE(Iterator i) is
  if i.t_stack.isEmpty() then
    BACKTRACKTOVAR(i)
  advanced ← False
  while i.node ≠ Null ∧ ¬advanced do
    while i.c_iter ≠ End ∧ ¬advanced do
      i.c_iter ← NEXTVARCHILD(i.node, i.c_iter)
      if i.c_iter ≠ End then
        x ← i.c_iter.var()
        t ← i.t_stack.top()
        s ← GOBBLEPREFIX(x, t)
        if s ≠ Null ∧
            (x.binding = Null ∨ x.binding = s) then
          i.t_stack.pop()
          for j ← t.num_args
              downto s.num_args + 1 do
            i.t_stack.push(t.args[j])
          if x.binding = Null then
            x.binding ← s
            i.t_proc.push((t, i.node, i.c_iter, True))
          else
            i.t_proc.push((t, i.node, i.c_iter, False))
          i.node ← i.node.child(x)
          advanced ← True
    t ← i.t_stack.top()
    if i.c_iter = End ∧ ¬t.head.isVar()
       ∧ D.child(t.head) ≠ Null then
      i.t_stack.pop()
      for j ← t.num_args downto 1 do
        i.t_stack.push(t.args[j])
      i.t_proc.push((t, i.node, End, False))
      i.node ← i.node.child(t.head)
      advanced ← True
    if ¬advanced then
      BACKTRACKTOVAR(i)
    else
      i.c_iter ← Start

procedure BACKTRACKTOVAR(Iterator i) is
  forever do
    if i.t_proc.isEmpty() then
      i.node ← Null
      return
    else
      (t, D, c_iter, var_unbound) ← i.t_proc.pop()
      label_arity ← i.node.label.type.arity
      t_arity ← t.type.arity
      for i ← 1 to label_arity − t_arity do
        i.t_stack.pop()
      i.t_stack.push(t)
      i.node ← D
      i.c_iter ← c_iter
      if var_unbound then
        i.node.label.binding ← Null
      if c_iter ≠ End then
        return
```

The pseudocode uses a slightly different representation of backtracking tuples than $\rightsquigarrow$. In the AdvanceX rule, σ changes only if the variable x was previously not bound. Instead of creating and storing substitutions explicitly, we simply remember whether the variable was bound in this step or not, in the *var_unbound* tuple component. Then we rely on

the label x of the current node and its *binding* field to carry the substitutions. Similarly, since our strategy is to traverse the tree by first visiting the variable-labeled child nodes, we need to remember how far we have come with this traversal. We store this information in the *c_iter* tuple component.

Fingerprint Indices Fingerprint indices [42] 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 immediately 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, and let $q < p$ denote that position q is a proper prefix of p (e.g., $\epsilon < 2 < 2.1$). Given a term t and a position p, the *fingerprint function* Gfpf is defined as

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

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

	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 $\mathcal{F}p(s) = (\mathsf{f}, \mathsf{a}, \mathsf{A}, \mathsf{N}, \mathsf{N}, \mathsf{B}, \mathsf{B})$ and $\mathcal{F}p(t) = (\mathsf{f}, \mathsf{g}, \mathsf{g}, \mathsf{A}, \mathsf{N}, \mathsf{a}, \mathsf{N})$. 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 f.g.g.A.N.a.N, together with other terms that share the same fingerprint. This scheme 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 $\{1,2\}$, but this is not graceful. Instead, it is 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: $t|^{\epsilon} = t$ and $\zeta\, t_n \ldots t_1\,|^{i.p} = t_i\,|^p$ 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 17 *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. The case $p = \epsilon$ is 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, $\sigma(s_1) = \sigma(t_1), \ldots, \sigma(s_i) = \sigma(t_i)$—i.e., $\sigma(s|^p) = \sigma(t|^p)$. □

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:

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

Except for the reversed numbering scheme, Gfpf' coincides with Gfpf 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 key difference between Gfpf and Gfpf' concerns 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 $s = x\,(\mathsf{f}\,\mathsf{b}\,\mathsf{c})$ and $t = \mathsf{g}\,\mathsf{a}\,(y\,\mathsf{d})$. Their fingerprints are $\mathcal{F}p'(s) = (\mathsf{A}, \mathsf{B}, \mathsf{f}, \mathsf{B}, \mathsf{B}, \mathsf{b}, \mathsf{c})$ and $\mathcal{F}p'(t) = (\mathsf{g}, \mathsf{a}, \mathsf{A}, \mathsf{N}, \mathsf{N}, \mathsf{B}, \mathsf{d})$. The terms are not unifiable due to the incompatibility at position 1.1 (c vs. d).

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

Lemma 18 *If terms s and t are unifiable, then $\mathit{Gfpf}'(s,p)$ and $\mathit{Gfpf}'(t,p)$ are compatible according to the unification matrix. If s generalizes t, then $\mathit{Gfpf}'(s,p)$ and $\mathit{Gfpf}'(t,p)$ are compatible according to the matching matrix.*

Proof We focus on the case of unification. By contraposition, it suffices to consider the eight blank cells in the unification matrix, where the rows correspond to $\mathit{Gfpf}'(s,p)$ and the columns correspond to $\mathit{Gfpf}'(t,p)$. Since unifiability is a symmetric relation, we can rule out four cases.

CASE f_1–f_2: By definition of Gfpf', $s|^p$ and $t|^p$ must be of the forms $\mathsf{f}_1\ \bar{s}$ and $\mathsf{f}_2\ \bar{t}$, respectively. Clearly, $s|^p$ and $t|^p$ are not unifiable. By Lemma 17, s and t are not unifiable.

CASE f_1–N, f_2–N, OR A–N: From $\mathit{Gfpf}'(t,p) = \mathsf{N}$, we deduce that $p \neq \epsilon$. Let $p = q.i.r$, where q is the longest prefix such that $\mathit{Gfpf}'(t,q) \neq \mathsf{N}$. Since $\mathit{Gfpf}'(t,q.i) = \mathsf{N}$, the head of $t|^q$ must be some symbol g. (For a variable head, we would have $\mathit{Gfpf}'(t,q.i) = \mathsf{B}$.) 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 17, s and t are not unifiable. □

Corollary 19 (Overapproximation) *If s and t are unifiable terms, then $\mathcal{F}p'(s)$ and $\mathcal{F}p'(t)$ are compatible according to the unification matrix. If s generalizes t, then $\mathcal{F}p'(s)$ and $\mathcal{F}p'(t)$ are compatible according to the matching matrix.*

Feature Vector Indices A clause C subsumes a clause D if there exists a substitution σ such that $\sigma(C) \subseteq D$. Subsumption is a crucial operation to prune the search space. Feature-vector indices [43] are an imperfect indexing data structure that can be used to retrieve clauses that subsume a query clause or that are subsumed by the query clause. 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.

6 Inference Rules

Saturating provers show the unsatisfiability of a clause set by systematically adding logical consequences, eventually deriving the empty clause as a witness of unsatisfiability. They implement two kinds of inference rules: *Generating rules* produce new clauses and are needed 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's main loop, which applies the rules, implements the given clause procedure [4]. The proof state is represented by two disjoint subsets of clauses, the set of *processed* clauses P and the set of *unprocessed* clauses U. Initially, all clauses are unprocessed. At each iteration of the 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. Simplification is performed on the given clause (using clauses in P as side premises), on clauses in P (using the given clause), and on newly generated clauses (again, using P).

Ehoh is based on the same logical calculus as E, except that it is generalized to λfHOL terms. The standard inference rules and completeness proof of superposition with respect to intensional Boolean-free λfHOL fragment of our logic can be reused verbatim; the only changes concern the basic definitions of terms and substitutions [10, Sect. 1]. Refutational completeness of superposition for λfHOL terms has been formally proved by Peltier [37] using Isabelle. We introduced support for first-class Boolean terms in Ehoh by extending the preprocessor, as explained in Sect. 8.

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

$$\frac{s \approx t \vee C \qquad u[s'] \not\approx v \vee D}{\sigma(u[t] \not\approx v \vee C \vee D)}\ \mathrm{SN} \qquad \frac{s \not\approx s' \vee C}{\sigma(C)}\ \mathrm{ER}$$

$$\frac{s \approx t \vee C \qquad u[s'] \approx v \vee D}{\sigma(u[t] \approx v \vee C \vee D)}\ \mathrm{SP} \qquad \frac{s \approx t \vee s' \approx u \vee C}{\sigma(t \not\approx u \vee s \approx u \vee C)}\ \mathrm{EF}$$

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

Equality resolution (ER) and equality factoring (EF) are single-premise rules that work on the entire left- or right-hand side of a literal of the given clause. To generalize them, it suffices to disable prefix optimization for unification.

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, the prover must be able to efficiently locate all relevant subterms, including λfHOL 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 unsuit-

able candidates. Thanks to prefix optimization, we can avoid polluting the index with all prefix subterms.

When 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.

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 storing clauses of the form $s \approx t$ indexed by s and t. When applying ES, 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 some arguments $\overline{u}$ remaining as unmatched. Checking for equality subsumption then amounts to checking that $v' = \sigma'(t)\ \overline{u}$, 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.

Pragmatic Extensions Since Ehoh is based on a monomorphic logic, the only way to support extensionality without changing the calculus is to add a set of extensionality axioms for every function type occurring in the problem [10, Sect. 3.1]. The evaluation by Bentkamp et al. of such an approach was discouraging [10, Sect. 6], so we decided to support extensionality via inference rules in Ehoh. We implemented two well-known incomplete rules we had experimented with in the context of Zipperposition.

The negative and positive extensionality (NE and PE) rules are defined as

$$\frac{s \not\approx t \vee C}{s\ (\mathsf{sk}\ \overline{x}) \not\approx t\ (\mathsf{sk}\ \overline{x}) \vee C}\ \text{NE} \qquad \frac{s\ x \approx t\ x \vee C}{s \approx t \vee C}\ \text{PE}$$

For NE, $\overline{x}$ contains all the variables occurring in s and t, the terms s and t are of function type, sk is a fresh Skolem symbol, and the literal $s \not\approx t$ is eligible for resolution [9, Sect. 5]. For PE, variable x does not occur in any of the s, t, or C, no literals are selected in C, and $s\ x \approx t\ x$ is a maximal literal.

Finally, we introduced an injectivity recognition (IR) rule, which detects injectivity axioms and asserts the existence of the inverse function for injective function symbols:

$$\frac{\mathsf{f}\ \overline{x}_n \not\approx \mathsf{f}\ \overline{y}_n \vee x_i \approx y_i}{\mathsf{sk}\ (\mathsf{f}\ \overline{x}_n)\ \overline{x}_J \approx x_i}\ \text{IR}$$

where sk is a fresh Skolem symbol, J is the largest subset of $\{1,\ldots,n\}$ such that $x_j = y_j$ for every $j \in J$. We denote the subsequence of $\overline{x}_n$ indexed by J by $\overline{x}_J$. Moreover, we require that $x_i \neq y_i$, all variables in $\overline{x}_K \cdot \overline{y}_K$ are distinct, where $K = \{1,\ldots,n\} \setminus J$, and neither $\overline{x}_K$ nor $\overline{y}_K$ shares variables with $\overline{x}_J$. For example, given $\mathsf{add}\ a\ b \not\approx \mathsf{add}\ a\ b' \vee a \approx b'$, IR can derive the existence of the inverse sk_1 characterized by $\mathsf{sk}_1\ (\mathsf{add}\ a\ b)\ a \approx b$.

7 Heuristics

E's heuristics are largely independent of the logic used and work unchanged for Ehoh. Yet, in preliminary experiments, we noticed that E proved some λfHOL benchmarks quickly using the applicative encoding (Sect. 1), whereas Ehoh timed out. There were enough such problems to prompt us to take a closer look. Based on these observations, we extended the heuristics to exploit λfHOL-specific features.

Term Order Generation The superposition calculus is 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. 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 $\mathsf{f}\ \mathsf{a}\ y \not\approx \mathsf{c}$ and $x\ \mathsf{b} \approx z$, where the first one is the negated conjecture.

Their applicative encoding is $@_{\iota_2,\iota_3}(@_{\iota_1,\iota_2\to\iota_3}(\mathsf{f},\mathsf{a}),y) \not\approx \mathsf{c}$ and $@_{\iota_2,\iota_3}(x,\mathsf{b}) \approx z$, where $@_{\tau,\upsilon}$ is a type-indexed family of 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, thereby 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 behavior observed on applicative 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 equal 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 of the same type.
- `typefreqrank` sorts the frequencies calculated by the function `typefreqcount` in increasing order and assigns each symbol a weight corresponding to its rank.
- `invtypefreqcount` is `typefreqcount`'s inverse. If `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.

We designed four more schemes (whose names begin with `comb` instead of `type`) that combine E's type-agnostic and Ehoh's type-aware approaches using a linear equation.

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 reflect the type information introduced by the applicative encoding, we implemented four type-aware precedence generation schemes. Ties are broken by comparing the symbols' number of occurrences and, if necessary, the position of their first occurrence in the input.

Literal Selection The side conditions of the superposition rules SN and SP (Sect. 6) rely on a literal selection function to restrict the set of *inference literals*, thereby reducing 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 is probably `SelectMaxLComplexAvoidPosPred`, which we abbreviate to `SelectMLCAPP`. It selects at most one negative literal, based on size, absence of variables, and maximality of the literal in the 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. On the other hand, 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 $max(L) = 1$ if L is a maximal literal of the clause it appears in; otherwise, $max(L) = 0$. Let $appvar(L) = 1$ if L is a literal where either side is an applied variable; otherwise, $appvar(L) = 0$. 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 $(max(L), 1 - appvar(L))$ according to the lexicographic order.
- `SelectMLCAPPPreferAppVar` selects a negative literal L with the maximal value of $(max(L), appvar(L))$ according to the lexicographic order.

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 order the clauses within a partition. E's main loop visits, in round-robin fashion, a set of priority queues. From a given queue, the clause with the highest priority and the smallest weight is selected. 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 eight parameters, notably *conj_mul*, a multiplier applied to the weight of conjecture symbols. This template works well for some applicatively encoded problems. Let $\mathsf{a} : \iota$, $\mathsf{f} : \iota \to \iota$, $x : \iota$, and $y : \iota \to \iota$, and consider the clauses $y\,x \not\approx x$ and $\mathsf{f}\,\mathsf{a} \approx \mathsf{a}$, where the first one is the negated conjecture. Their encoding is $@_{\iota,\iota}(y,x) \not\approx x$ and $@_{\iota,\iota}(\mathsf{f},\mathsf{a}) \approx \mathsf{a}$. The encoded clauses share $@_{\iota,\iota}$, whose weight will be multiplied by *conj_mul*—usually 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, we coded a new type-aware template, `CRSTypeWeight`, that 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*.

Natively supporting λfHOL allows the prover to recognize applied variables. It may make sense to extend clause

weight templates to either penalize or promote clauses with such variables. To support this extension, we added the following parameter to `CRSTypeWeight`, as well as to some other E's weight function templates: *appv_mul* is a multiplier applied to terms $s = x\,\bar{t}_n$, where s is either side of the literal and $n > 0$. 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.

Configurations and Modes A combination of parameters, including term order, literal selection, and clause selection, is called a *configuration*. For years, E has provided an *auto* mode that 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 that applies a portfolio of configurations in sequence on the given problem, restarting the prover for each configuration.

Configurations that are suitable for 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 the queue will be visited n times in the round-robin scheme before moving to the next one. The first argument to each template is the clause priority scheme.

8 Preprocessing

E's preprocessor transforms first-order formulas into clausal normal form, before the main loop is started. Since literals of clauses are (dis)equations, E encodes nonequational literals such as $\mathsf{even}(n)$ as equations $\mathsf{even}(n) \approx \mathsf{T}$. 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 both 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 [34, Sect. 6] is to ensure that Skolem symbols are always applied to a minimum number of 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 plan 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 $\mathsf{f}\,\bar{x}_m \approx t$, where the variables $\bar{x}_m$ 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 $\{y \mapsto \lambda x.\ x\}$ instance and β-conversion.) For the moment, we have simply disabled definition unfolding in Ehoh. We will enable it again once we have added support for λ-terms.

Higher-order logic treats formulas as terms of Boolean type, erasing the distinction between terms and formulas. As a consequence, formulas might appear as arguments not only to logical connectives but also to function symbols or applied variables—e.g., $\mathsf{p}\,(\mathsf{a} \wedge \mathsf{b})$, $y\,(\neg\,\mathsf{a})$. We call such formulas *nested*. Kotelnikov et al. [26] describe a modification to Vampire's clausification algorithm to support nested formulas. We adapt their approach to the clausification algorithm [35] used by E. Given a formula φ to clausify, the following procedure removes nested formulas:

1. Let $\chi = \varphi|_p$ be the leftmost outermost nested formula that is different from T, $\bot$, or a variable x, if one exists; otherwise, skip to step 2. Let $p = q.r$ where q is the longest strict prefix of p such that $\psi = \varphi|_q$ is a formula. Let $\psi' = (\chi \to \psi[\mathsf{T}]_r) \wedge (\neg\chi \to \psi[\bot]_r)$. Replace φ by $\varphi[\psi']_q$ and repeat this step.
2. Apply all the steps of E's clausification algorithm up to and including skolemization.
3. Skolemization might replace Boolean variables by new terms with predicate symbol heads. To remove them, follow step 1.

4. Perform the remaining steps of E's clausification algorithm, resulting in a set of clauses.
5. Let C be a clause that contains a literal L of the form $x \approx \mathsf{T}$ or $x \not\approx \mathsf{T}$, where x is a Boolean variable, if one exists; otherwise, terminate. Delete C if it also contains the complement of L. Otherwise, replace C with the clause $C[x \mapsto \bot]$ if L is of the form $x \approx \mathsf{T}$ and else $C[x \mapsto \mathsf{T}]$. Trivial literals $\bot \approx \mathsf{T}$ and $\mathsf{T} \not\approx \mathsf{T}$ are removed from the resulting clause. Repeat this step.

As an example, consider the formula $\mathsf{f}\ x \approx x \to \mathsf{p}\ (\mathsf{a} \wedge \mathsf{b})$. Step 1 moves the subterm $\mathsf{a} \wedge \mathsf{b}$ outward, yielding $\mathsf{f}\ x \approx x \to ((\mathsf{a} \wedge \mathsf{b}) \to \mathsf{p}\ \mathsf{T}) \wedge (\neg(\mathsf{a} \wedge \mathsf{b}) \to \mathsf{p}\ \bot)$. This formula can be clausified further as usual.

Theorem 20 (Total Correctness) *The above procedure always terminates and produces a set of clauses that is equisatisfiable with the original formula φ in λfHOL with interpreted Booleans and that contains no nested formulas other than T, $\bot$, and variables.*

Proof It is easy to see that steps 1, 3, and 5 produce equivalent formulas or clauses. Moreover, steps 1 and 3 remove all offending nested formulas (i.e., other than T, $\bot$, and variables). In conjunction with the standard clausification algorithm, which preserves and reflects satisfiability, our procedure gives correct results when it terminates.

To prove termination, we will use a measure function $\mathcal{W}$ to natural numbers that decreases with each application of step 1 or 3. Steps 2 and 4 rely on a terminating algorithm, whereas each application of step 5 decreases the size of a clause. We define $\mathcal{W}$ by $\mathcal{W}(\forall x.\ s) = \mathcal{W}(\exists x.\ s) = \mathcal{W}(s)$; $\mathcal{W}(\zeta\, \overline{s}_n) = \sum_{i=1}^{n} \mathcal{W}(s_i)$ if ζ is a logical connective (including T and $\bot$); and $\mathcal{W}(\zeta\, \overline{s}_n) = 3^k(1 + \sum_{i=1}^{n} \mathcal{W}(s_i))$ otherwise, where k is the number of offending outermost nested formulas in $\zeta\, \overline{s}_n$. We must show $\mathcal{W}(\psi) > \mathcal{W}(\psi')$. By definition, ψ is of the form $\zeta\, \overline{s}_n$, where ζ is not a logical connective. Thus $\mathcal{W}(\psi) = 3^k(1 + \sum_{i=1}^{n} \mathcal{W}(s_i))$. Steps 1 and 3 substitute T or $\bot$, of measure 0, for a nested formula χ (including χ's own nested formulas) in ψ. Clearly, the longer r is, the more $\mathcal{W}(\psi')$ decreases. Taking $|r| = 1$, we get the upper bound $2\mathcal{W}(\chi) + 2 \cdot 3^{k-1}(1 + \sum_{i=1}^{n} \mathcal{W}(s_i) - \mathcal{W}(\chi))$ for $\mathcal{W}(\psi')$, which is less than $\mathcal{W}(\psi) = 3^k(1 + \sum_{i=1}^{n} \mathcal{W}(s_i))$. □

The output may contain T, $\bot$, or Boolean variables as nested formulas. Since E was first developed as an untyped prover, unification of a variable with a Boolean constant was disallowed to avoid unsoundness. We needed to undo this in Ehoh. Ehoh must also remove trivial literals $\bot \approx \mathsf{T}$ and $\mathsf{T} \not\approx \mathsf{T}$ that emerge during proof search.

9 Evaluation

How useful are Ehoh's new heuristics? And how does Ehoh perform compared with E, used directly or in tandem 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. For the second question, we compared Ehoh's success rate and speed on λfHOL problems with native higher-order provers and on applicatively encoded problems with E. We also included first-order benchmarks to measure Ehoh's overhead.

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 [50]. The experiments were performed on StarExec [47] nodes equipped with Intel Xeon E5-2609 0 CPUs clocked at 2.40 GHz.

	`f`	`if`	`tf`	`itf`	`cmf`	`icmf`
`fcn`	2294	2288	2287	2297	2290	2287
`ifcn`	2371	2373	2374	2370	2369	2377
`fr`	2326	2317	2323	2329	2322	2318
`ifr`	2383	<u>2379</u>	2376	2380	2381	2381
`tfcn`	2305	2314	2301	2306	2302	2311
`itfcn`	2386	2381	2389	2388	2384	2379
`tfr`	2326	2334	2322	2334	2321	2336
`itfr`	2390	2382	2390	**2394**	2387	2386
`cmfcn`	2273	2281	2271	2285	2269	2280
`icmfcn`	2380	2375	2382	2379	2380	2375
`cmfr`	2321	2313	2319	2321	2318	2312
`icmfr`	2368	2378	2371	2378	2368	2380

Fig. 1 Evaluation of weight and precedence generation schemes

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 affects overall performance. All heuristic parameters were tested on a 5012 problem suite generated using Sledgehammer, consisting of four variants of the Judgment Day [17] suite. The problems were given in native λfHOL syntax. The experiments described in this subsection were carried out using an earlier E version (2.3).

Evaluating the new weight and precedence generation heuristics amounted to testing each possible combination of frequency-based schemes, including E's original type-agnostic schemes. Figure 1 shows the number of solved (i.e., proved or disproved) problems for each combination. In this and the following figures, the underlined number is for *boa*, whereas bold singles out the best value. In the names of the generation schemes, we abbreviated `inv` to `i`, `type` to `t`, `freq` to `f`, `comb` to `cm`, `count` to `cn`, and `rank` to `r`.

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 appear to be less efficient than the type-aware schemes.

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

Fig. 2 Evaluation of weight function and *appv_mult* factor

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

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

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.

To test the effect of the new type-based weight function, we replaced *boa*'s queue, which uses `4.CRSWeight(...)`, with the queue ordered by `4.CRSTypeWeight(...)`. We call the original heuristic `W` and the type-aware alternative `TW`. We chose nine values for testing the effect of the applied variable multiplier *appv_mult*. Figure 2 summarizes the results of combining `W` or `TW` with the different *appv_mult* values. Applying a multiplier smaller than 1, which corresponds to preferring literals containing applied variables, can lose dozens of solutions. Overall, using the type-aware heuristic seems slightly detrimental.

Finally, we evaluated the new clause priority function `ByAppVarNum`, by replacing `4.CRSWeight(ConstPrio, ...)` with `4.CRSWeight(ByAppVarNum,...)` in *boa*'s specification. `ConstPrio` assigns each clause the same priority. The results are inconclusive.

The results presented above give an idea 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`.

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.

Main Evaluation We now present a more detailed evaluation of *hoboa*, along with other configurations, on a larger benchmark suite. Our raw data are publicly available.[2]

The benchmarks are divided into four sets: (1) 1147 first-order TPTP [51] problems belonging to the FOF (untyped) and TF0 (monomorphic) categories, excluding arithmetic; (2) 5012 Sledgehammer-generated problems from the Judgment Day [17] suite, targeting the monomorphic first-order logic embodied by TPTP TF0; (3) all 955 monomorphic higher-order problems from the TH0 category of the TPTP belonging to our extension of λfHOL; (4) 5012 Judgment Day problems targeting the λfHOL fragment of TPTP TH0.

The TPTP includes benchmarks from various areas of computer science and mathematics. It is the de facto standard for evaluating automatic provers, but it has few higher-order problems. For the first group of benchmarks, we randomly selected 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 (SH32 and SH512), to represent both small and large problems. Each subgroup consists of two sub-subgroups of 1253 problems, generated by using either λ-lifting or SK-style combinators to encode λ-expressions.

To ascertain the effectiveness of our approach, we evaluated Ehoh against E used on applicative encodings of problems (denoted by @+E). For reference, we also evaluated the latest versions of higher-order provers that competed in the THF division of the 2019 edition of CASC [52]: CVC4 1.8 prerelease [6], Leo-III 1.4 [46], Satallax 3.4 [18], Vampire 4.4 [14], and Zipperposition 1.6 [9]. Like at CASC, we used different versions of Vampire for first-order and higher-order problems. Similarly, Zipperposition does not use E as backend when it is run on first-order problems and uses different heuristics on first- and higher-order problems. The genuine higher-order provers have the unfair advantage that they can instantiate higher-order variables with λ-terms. Thus, some formulas that are provable by these systems may be nontheorems for @+E and Ehoh, or they may require tedious reasoning about λ-lifted functions or SK-style combinators. An example is the conjecture $\exists f.\, \forall x\, y.\, f\, x\, y \approx \mathrm{g}\, y\, x$, whose proof requires taking $\lambda x\, y.\, \mathrm{g}\, y\, x$ as the witness for f.

We ran all provers except Satallax (which only supports THF) on first-order benchmarks to measure the overhead introduced by our extensions, as well as that entailed by the applicative encoding. Figure 4 gives the number of problems each system proved. In each column, bold highlights

[2] `https://doi.org/10.5281/zenodo.4045452`

	First-order			Higher-order		
	TPTP	SH32	SH512	TPTP	SH32	SH512
E a	624	938	1237			
E as	**665**	**957**	**1298**			
E b	550	943	1242			
@+E a	531	932	1111	686	952	1125
@+E as	571	949	1148	692	969	1164
@+E b	536	943	1227	690	959	1267
Ehoh a	624	939	1236	694	966	1235
Ehoh as	**665**	**957**	1296	**699**	**988**	**1309**
Ehoh b	550	943	1242	697	967	1262
Ehoh hb	504	947	1231	693	975	1267
CVC4	567	956	1361	745	973	**1351**
Leo-III	548	960	1239	**834**	967	1266
Vampire	**728**	**968**	**1401**	805	979	1214
Satallax				827	871	1019
Zipperposition	496	933	1187	815	976	1069

Fig. 4 Number of proved problems

	First-order			Higher-order		
	TPTP	SH32	SH512	TPTP	SH32	SH512
E a	0.22	0.15	0.54			
E as	0.38	0.20	0.74			
E b	0.43	**0.07**	0.56			
@+E a	0.61	0.18	**0.38**	0.03	0.21	**0.32**
@+E as	0.91	0.18	0.39	0.06	0.25	0.33
@+E b	0.53	0.12	0.81	0.09	0.20	0.54
Ehoh a	**0.21**	0.15	0.54	0.03	0.08	0.51
Ehoh as	0.38	0.20	0.73	0.07	0.14	0.60
Ehoh b	0.42	**0.07**	0.58	**0.02**	**0.07**	0.37
Ehoh hb	0.69	0.12	1.06	0.10	0.13	0.56
CVC4	3.02	1.58	1.75	1.22	2.44	1.65
Leo-III	1.33	0.52	5.63	0.49	0.89	6.54
Vampire	0.67	0.43	1.50	0.76	1.89	4.84
Satallax				2.45	5.22	10.12
Zipperposition	3.81	1.60	5.09	0.76	2.21	6.31

Fig. 5 Average running times on the problems proved by all systems

the best E value and the best value overall. We considered the E modes *auto* (a) and *autoschedule* (as) and the configurations *boa* (b) and *hoboa* (hb).

We observe the following. First, comparing the Ehoh row with the E row, we see that Ehoh's overhead is barely noticeable—the difference is at most two problems. Second, Ehoh outperforms the applicative encoding on both first-order and higher-order problems. Nevertheless, the raw evaluation data reveal that there are quite a few higher-order problems that @+E proves faster than Ehoh. Third, it is advantageous to use the higher-order versions of the Sledgehammer problems, although the difference in success rate is small, especially for SH512. Fourth, the new *hoboa* 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. Fifth, Ehoh cannot compete against the best higher-order systems, but this is no surprise given that it does not yet support λ-expressions and higher-order unification.

Next to the success rate, the time in which a prover gives an answer is also an important consideration. Figure 5 compares the average running times, in seconds, of the various systems on the problems that all of the applicable systems proved. Clearly, Ehoh incurs little overhead on first-order problems. The raw evaluation data reveal that for *boa*, it takes Ehoh 2747 s to prove all first-order problems that E, @+E, and Ehoh can all prove using this configuration, compared with 2728 s for E, amounting to a 0.7% overhead. For comparison, @+E needs 3939 s—a 44% overhead.

10 Discussion and Related Work

Our working hypothesis is that it is possible to extend 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 [9, 10, 56]; on the practical side, we are implementing such extensions in a state-of-the-art prover.

The work described in this article required modifying many parts of the E prover. The invariant that variables cannot be applied and that symbols are always passed the same number of arguments were entrenched in E's code, 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.

Traditionally, most higher-order provers were designed from the ground up to target higher-order logic. Two exceptions are Otter-λ by Beeson [8] and Zipperposition by Cruanes et al. [9, 20]. Otter-λ adds λ-terms and second-order unification to the superposition prover Otter [31]. Zipperposition, also based on superposition, was extended to Boolean-free higher-order logic by Bentkamp et al. [9]. Its performance is a far cry from E's, but it is easier to modify. Vukmirović et al. also used it to test and evaluate higher-order unification procedures [54] and Boolean reasoning [56]. Zipperposition now includes Ehoh as a backend in a cooperative architecture. Finally, there is recent work by the developers of Vampire [14] and of the SMT (satisfiability modulo theories) solvers CVC4 and veriT [6] to extend their provers to higher-order logic.

Native higher-order reasoning was pioneered by Robinson [39], Andrews [1], and Huet [24]. Andrews [2] and Benzmüller and Miller [12] provide excellent surveys. TPS, by Andrews et al. [3], was based on expansion proofs and lets users specify proof outlines. The Leo family of systems, developed by Benzmüller and his colleagues, is based on resolution and paramodulation. LEO [11] supported extensionality on the calculus level and introduced the cooperative paradigm to integrate first-order provers. Leo-III [46] expands the cooperation with SMT solvers and introduces

term orders in a pragmatic, incomplete way. Brown's Satallax [18] is based on a complete higher-order tableau calculus, guided by a SAT solver; later versions also cooperate with E and Ehoh. Another noteworthy system is Lindblad's agsyHOL [28]. 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 via a translation. Robinson [40] outlined this approach decades before tools such as MizAR [53], Sledgehammer [36], HOLyHammer [25], and CoqHammer [22] popularized it in proof assistants. In addition to performing an applicative encoding, such translations must eliminate the λ-expressions [21, 33] and encode the type information [15]. In practice, on problems with a large first-order component, translations perform very well compared with the existing native provers [48]. Largely thanks to Sledgehammer, Isabelle often came in close second at CASC, even defeating Satallax in 2012 [49].

By removing the need for the applicative encoding, our work reduces the translation gap. The encoding buries the λfHOL terms' heads under layers of @ symbols. Terms double in size, cluttering the data structures, and twice as many subterm positions must be considered for inferences. Moreover, the encoding is incompatible with interpreted operators, notably for arithmetic. A common remedy is to introduce proxies to connect an uninterpreted nullary symbol with its interpreted counterpart (e.g., $@(@(\mathsf{add}, x), y) \approx x + y$), but this is clumsy. A further complication is that in a monomorphic logic, @ is not a single symbol but a family of symbols $@_{\tau,\upsilon}$, which must be correctly introduced and recognized. Finally, the encoding must be undone in the proofs. 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 λ-free higher-order problems and very little overhead for first-order problems, as we would expect from a graceful generalization.

Despite its lack of support for λ-terms, Ehoh is already deployed as a backend in the leading higher-order provers Satallax and Zipperposition. Ehoh will also form the basis of our work toward stronger higher-order automation. Our aim is to turn it into a prover that excels on proof obligations arising in interactive verification, which tend to be large but only mildly higher-order [48]. The next steps will be to extend Ehoh's data structures with λ-expressions and implement the higher-order unification procedure by Vukmirović et al. [54]. These techniques are cornerstones of our prototype Zipperposition, which dominated the higher-order proving division of the 2020 edition of CASC.

Acknowledgments 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, Dmitriy Traytel, and the anonymous reviewers for suggesting many improvements to this text. 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. Finally, we thank the TACAS 2019 chairs and editors of this special issue, Tomáš Vojnar and Lijun Zhang, for their patience with us.

Funding 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). Blanchette has received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward). He also benefited from the NWO Incidental Financial Support scheme.

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: J.A. Robinson, A. Voronkov (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. Avenhaus, J., Denzinger, J., Fuchs, M.: DISCOUNT: A system for distributed equational deduction. In: J. Hsiang (ed.) RTA-95, *LNCS*, vol. 914, pp. 397–402. Springer (1995)
5. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
6. Barbosa, H., Reynolds, A., Ouraoui, D.E., Tinelli, C., Barrett, C.W.: Extending SMT solvers to higher-order logic. In: P. Fontaine (ed.) CADE-27, *LNCS*, vol. 11716, pp. 35–54. Springer (2019)
7. Becker, H., Blanchette, J.C., Waldmann, U., Wand, D.: A transfinite Knuth–Bendix order for lambda-free higher-order terms. In: L. de Moura (ed.) CADE-26, *LNCS*, vol. 10395, pp. 432–453. Springer (2017)
8. Beeson, M.: Lambda logic. In: D.A. Basin, M. Rusinowitch (eds.) IJCAR 2004, *LNCS*, vol. 3097, pp. 460–474. Springer (2004)
9. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas. In: P. Fontaine (ed.) CADE-27, *LNCS*, vol. 11716, pp. 55–73. Springer (2019)
10. Bentkamp, A., Blanchette, J.C., Cruanes, S., Waldmann, U.: Superposition for lambda-free higher-order logic. In: D. Galmiche, S. Schulz, R. Sebastiani (eds.) IJCAR 2018, *LNCS*, vol. 10900, pp. 28–46. Springer (2018)
11. Benzmüller, C., Kohlhase, M.: System description: LEO—a higher-order theorem prover. In: C. Kirchner, H. Kirchner (eds.) CADE-15, *LNCS*, vol. 1421, pp. 139–144. Springer (1998)

12. Benzmüller, C., Miller, D.: Automation of higher-order logic. In: J.H. Siekmann (ed.) Computational Logic, *Handbook of the History of Logic*, vol. 9, pp. 215–254. Elsevier (2014)
13. Benzmüller, C., Sultana, N., Paulson, L.C., Theiss, F.: The higher-order prover LEO-II. J. Autom. Reason. **55**(4), 389–404 (2015)
14. Bhayat, A., Reger, G.: Restricted combinatory unification. In: P. Fontaine (ed.) CADE-27, *LNCS*, vol. 11716, pp. 74–93. Springer (2019)
15. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. Log. Meth. Comput. Sci. **12**(4), 13:1–13:52 (2016)
16. Blanchette, J.C., Waldmann, U., Wand, D.: A lambda-free higher-order recursive path order. In: J. Esparza, A.S. Murawski (eds.) FoSSaCS 2017, *LNCS*, vol. 10203, pp. 461–479. Springer (2017)
17. Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: J. Giesl, R. Hähnle (eds.) IJCAR 2010, *LNCS*, vol. 6173, pp. 107–121. Springer (2010)
18. Brown, C.E.: Satallax: An automatic higher-order prover. In: B. Gramlich, D. Miller, U. Sattler (eds.) IJCAR 2012, *LNCS*, vol. 7364, pp. 111–117. Springer (2012)
19. Cruanes, S.: Extending superposition with integer arithmetic, structural induction, and beyond. PhD thesis, École polytechnique (2015)
20. Cruanes, S.: Superposition with structural induction. In: C. Dixon, M. Finger (eds.) FroCoS 2017, *LNCS*, vol. 10483, pp. 172–188. Springer (2017)
21. Czajka, Ł.: Improving automation in interactive theorem provers by efficient encoding of lambda-abstractions. In: J. Avigad, A. Chlipala (eds.) CPP 2016, pp. 49–57. ACM (2016)
22. Czajka, Ł., Kaliszyk, C.: Hammer for Coq: Automation for dependent type theory (2018)
23. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: M. Felleisen, P. Gardner (eds.) ESOP 2013, *LNCS*, vol. 7792, pp. 125–128. Springer (2013)
24. Huet, G.P.: A mechanization of type theory. In: N.J. Nilsson (ed.) IJCAI-73, pp. 139–146. William Kaufmann (1973)
25. Kaliszyk, C., Urban, J.: HOL(y)Hammer: Online ATP service for HOL Light. Math. Comput. Sci. **9**(1), 5–22 (2015)
26. Kotelnikov, E., Kovács, L., Suda, M., Voronkov, A.: A clausal normal form translation for FOOL. In: C. Benzmüller, G. Sutcliffe, R. Rojas (eds.) GCAI 2016, *EPiC Series in Computing*, vol. 41, pp. 53–71. EasyChair (2016)
27. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: N. Sharygina, H. Veith (eds.) CAV 2013, *LNCS*, vol. 8044, pp. 1–35. Springer (2013)
28. Lindblad, F.: A focused sequent calculus for higher-order logic. In: S. Demri, D. Kapur, C. Weidenbach (eds.) IJCAR 2014, *LNCS*, vol. 8562, pp. 61–75. Springer (2014)
29. Löchner, B.: Things to know when implementing KBO. J. Autom. Reason. **36**(4), 289–310 (2006)
30. Löchner, B., Schulz, S.: An evaluation of shared rewriting. In: H. de Nivelle, S. Schulz (eds.) IWIL-2001, pp. 33–48. Max-Planck-Institut für Informatik (2001)
31. McCune, W.: OTTER 2.0. In: M.E. Stickel (ed.) CADE-10, *LNCS*, vol. 449, pp. 663–664. Springer (1990)
32. McCune, W.: Experiments with discrimination-tree indexing and path indexing for term retrieval. J. Autom. Reason. **9**(2), 147–167 (1992)
33. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Autom. Reason. **40**(1), 35–60 (2008)
34. Miller, D.A.: A compact representation of proofs. Stud. Log. **46**(4), 347–370 (1987)
35. Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: J.A. Robinson, A. Voronkov (eds.) Handbook of Automated Reasoning (in 2 volumes), pp. 335–367. Elsevier and MIT Press (2001)
36. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: G. Sutcliffe, S. Schulz, E. Ternovska (eds.) IWIL-2010, *EPiC*, vol. 2, pp. 1–11. EasyChair (2012)
37. Peltier, N.: A variant of the superposition calculus. Archive of Formal Proofs (2016). `https://www.isa-afp.org/`
38. Reger, G., Suda, M.: Checkable proofs for first-order theorem proving. In: G. Reger, D. Traytel (eds.) ARCADE 2017, *EPiC Series in Computing*, vol. 51, pp. 55–63. EasyChair (2017)
39. Robinson, J.: Mechanizing higher order logic. In: B. Meltzer, D. Michie (eds.) Machine Intelligence, vol. 4, pp. 151–170. Edinburgh University Press (1969)
40. Robinson, J.: A note on mechanizing higher order logic. In: B. Meltzer, D. Michie (eds.) Machine Intelligence, vol. 5, pp. 121–135. Edinburgh University Press (1970)
41. Schulz, S.: E—a brainiac theorem prover. AI Commun. **15**(2–3), 111–126 (2002)
42. Schulz, S.: Fingerprint indexing for paramodulation and rewriting. In: B. Gramlich, D. Miller, U. Sattler (eds.) IJCAR 2012, *LNCS*, vol. 7364, pp. 477–483. Springer (2012)
43. Schulz, S.: Simple and efficient clause subsumption with feature vector indexing. In: M.P. Bonacina, M.E. Stickel (eds.) Automated Reasoning and Mathematics—Essays in Memory of William W. McCune, *LNCS*, vol. 7788, pp. 45–67. Springer (2013)
44. Schulz, S.: We know (nearly) nothing! But can we learn? In: G. Reger, D. Traytel (eds.) ARCADE 2017, *EPiC Series in Computing*, vol. 51, pp. 29–32. EasyChair (2017)
45. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: P. Fontaine (ed.) CADE-27, *LNCS*, vol. 11716, pp. 495–507. Springer (2019)
46. Steen, A., Benzmüller, C.: The higher-order prover Leo-III. In: D. Galmiche, S. Schulz, R. Sebastiani (eds.) IJCAR 2018, *LNCS*, vol. 10900, pp. 108–116. Springer (2018)
47. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: S. Demri, D. Kapur, C. Weidenbach (eds.) IJCAR 2014, *LNCS*, vol. 8562, pp. 367–373. Springer (2014)
48. Sultana, N., Blanchette, J.C., Paulson, L.C.: LEO-II and Satallax on the Sledgehammer test bench. J. Appl. Log. **11**(1), 91–102 (2013)
49. Sutcliffe, G.: The 6th IJCAR automated theorem proving system competition—CASC-J6. AI Comm. **26**(2), 211–223 (2013)
50. Sutcliffe, G.: The CADE-26 automated theorem proving system competition—CASC-26. AI Commun. **30**(6), 419–432 (2017)
51. 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)
52. Sutcliffe, G.: The CADE-27 automated theorem proving system competition—CASC-27. AI Commun. **32**(5-6), 373–389 (2019)
53. Urban, J., Rudnicki, P., Sutcliffe, G.: ATP and presentation service for Mizar formalizations. J. Autom. Reason. **50**(2), 229–241 (2013)
54. Vukmirović, P., Bentkamp, A., Nummelin, V.: Efficient full higher-order unification. In: Z.M. Ariola (ed.) FSCD 2020, *LIPIcs*, vol. 167, pp. 5:1–5:17. Schloss Dagstuhl (2020)
55. Vukmirović, P., Blanchette, J.C., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic. In: T. Vojnar, L. Zhang (eds.) TACAS 2019, *LNCS*, vol. 11427, pp. 192–210. Springer (2019)
56. Vukmirović, P., Nummelin, V.: Boolean reasoning in a higher-order superposition prover. In: P. Fontaine, K. Korovin, I.S. Kotsireas, P. Rümmer, S. Tourret (eds.) PAAR+SC-Square 2020, *CEUR Workshop Proceedings*, vol. 2752, pp. 148–166. CEUR-WS.org (2020)
57. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: R.A. Schmidt (ed.) CADE-22, *LNCS*, vol. 5663, pp. 140–145. Springer (2009)