

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

Petar Vukmirović, Jasmin Christian Blanchette, Simon Cruanes, Stephan Schulz

▶ **To cite this version:**

Petar Vukmirović, Jasmin Christian Blanchette, Simon Cruanes, Stephan Schulz. Extending a Brainiac Prover to Lambda-Free Higher-Order Logic. TACAS 2019 - 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Apr 2019, Prague, Czech Republic. pp.192-210. hal-02178274

HAL Id: hal-02178274
https://inria.hal.science/hal-02178274

Submitted on 9 Jul 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

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

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

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

1 Introduction

Superpostition-based provers, such as E [26], SPASS [33], and Vampire [18], are among the most successful first-order reasoning systems. They serve as backends in various frameworks, including software verifiers (Why3 [15]), automatic higher-order theorem provers (Leo-III [27], Satallax [12]), and "hammers" in proof assistants (HOLyHammer for HOL Light [17], Sledgehammer for Isabelle [21]). Decades of research have gone into refining calculi, devising efficient data structures and algorithms, and developing heuristics to guide proof search. This work has mostly focused on first-order logic with equality, with or without arithmetic.

Research on higher-order automatic provers has resulted in systems such as LEO [8], Leo-II [9], and Leo-III [27], based on resolution and paramodulation, and Satallax [12], based on tableaux. These provers feature a "cooperative" architecture, pioneered by LEO: They are full-fledged higher-order provers that regularly invoke an external first-order prover in an attempt to finish the proof quickly using only first-order reasoning. However, the first-order backend will succeed only if all the necessary higher-order reasoning has been performed, meaning that much of the first-order reasoning is carried out by the slower higher-order prover. As a result, this architecture leads to suboptimal performance on first-order problems and on problems with a large first-order component. For example, at the 2017 installment of the CADE ATP System Competition (CASC) [30], Leo-III, using E

as one of its backends, proved 652 out of 2000 first-order problems in the Sledgehammer division, compared with 1185 for E on its own and 1433 for Vampire.

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

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

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

Our vehicle is E, a prover developed primarily by Schulz. It is written in C and offers good performance, with the emphasis on "brainiac" heuristics rather than raw speed. E regularly scores among the top systems at CASC, and usually is the strongest open source[1] prover in the relevant divisions. It also serves as a backend for competitive higher-order provers. We refer to our extended version of E as Ehoh. It corresponds to E version 2.3 configured with `-enable-ho`. A prototype of Ehoh is described in Vukmirović's MSc thesis [31].

The three main challenges are generalizing the term representation (Sect. 3), the unification algorithm (Sect. 4), and the indexing data structures (Sect. 5). We also adapted the inference rules (Sect. 6) and the heuristics (Sect. 7). This paper explains the key ideas. Details, including correctness proofs, are given in a separate technical report [32].

A novel aspect of our work is *prefix optimization*. Higher-order terms contain twice as many proper subterms as first-order terms; for example, the term f (g a) b contains not only the argument subterms g a, a, b but also the "prefix" subterms f, f (g a), g. Using prefix optimization, the prover traverses subterms recursively in a first-order fashion, considering all the prefixes of the current subterm together, at no significant additional cost. Our experiments (Sect. 8) show that Ehoh is effectively as fast as E on first-order problems and can also prove higher-order problems that do not require synthesizing λ-terms. As a next step, we plan to add support for λ-terms and higher-order unification.

2 Logic

Our logic corresponds to the intensional λ-free higher-order logic (λfHOL) described by Bentkamp, Blanchette, Cruanes, and Waldmann [7, Sect. 2]. Another possible name for this logic would be "applicative first-order logic." Extensionality can be obtained by adding suitable axioms [7, Sect. 3.1].

A type is either an atomic type ι or a function type $\tau \to \upsilon$, where τ and υ are themselves types. Terms, ranged over by s, t, u, v, are either *variables* $x, y, z, \ldots$,

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

(*function*) *symbols* $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{f}, \mathsf{g}, \ldots$ (often called "constants" in the higher-order literature), or binary applications $s\ t$. Application associates to the left, whereas $\to$ associates to the right. The typing rules are as for the simply typed λ-calculus. A term's *arity* is the number of extra arguments it can take; thus, if f has type $\iota \to \iota \to \iota$ and a has type ι, then f is binary, $\mathsf{f}\ \mathsf{a}$ is unary, and $\mathsf{f}\ \mathsf{a}\ \mathsf{a}$ is nullary. Terms have a unique "flattened" decomposition of the form $\zeta\ s_1\ \ldots\ s_m$, where ζ, the *head*, is a variable x or symbol f. We abbreviate tuples $(a_1, \ldots, a_m)$ to $\overline{a_m}$ or $\overline{a}$; abusing notation, we write $\zeta\ \overline{s_m}$ for the curried application $\zeta\ s_1\ \ldots\ s_m$.

An equation $s \approx t$ corresponds to an unordered pair of terms. A literal L is an equation or its negation. 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.

A well-known technique to support λfHOL using first-order reasoning systems is to employ the *applicative encoding*. Following this scheme, every n-ary symbol is converted to a nullary symbol, and application is represented by a distinguished binary symbol @. For example, the λfHOL term $\mathsf{f}\ (x\ \mathsf{a})\ \mathsf{b}$ is encoded as the first-order term $@(@(\mathsf{f}, @(x, \mathsf{a})), \mathsf{b})$. However, this representation is not graceful; it clutters data structures and impacts proof search in subtle ways, leading to poorer performance, especially on large benchmarks. In our empirical evaluation, we find that for some prover modes, the applicative encoding incurs a 15% decrease in success rate (Sect. 8). For these and further reasons (Sect. 9), it is not an ideal basis for higher-order reasoning.

3 Types and Terms

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

Types. For most of its history, E supported only untyped first-order logic. Cruanes implemented support for atomic types for E 2.0 [13, p. 117]. Symbols f are declared with a type signature: $\mathsf{f} : \tau_1 \times \cdots \times \tau_m \to \tau$. Atomic types are represented by integers in memory, leading to efficient type comparisons.

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

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

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

In higher-order logic, variables may have function type and be applied, and symbols can be applied to fewer arguments than specified by their type signatures. A natural representation of λfHOL terms as tagged unions would distinguish between variables x, symbols f, and binary applications $s\ t$. However, this scheme suffers from memory fragmentation and linear-time access, as with the representation of types, affecting performance on purely or mostly first-order problems. Instead, we propose a flattened representation, as a generalization of E's existing data structures: Allow arguments to variables, and for symbols let `arity` be the number of *actual* arguments.

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 of this technique is that some normalization is necessary after substitution: Whenever a variable is instantiated by a term with a symbol head, the @ symbol must be eliminated. Applying the substitution $\{x \mapsto \mathsf{f}\ \mathsf{a}\}$ to the cell $@(x, \mathsf{b}, \mathsf{c})$ must produce the cell $\mathsf{f}(\mathsf{a}, \mathsf{b}, \mathsf{c})$ and not $@(\mathsf{f}(\mathsf{a}), \mathsf{b}, \mathsf{c})$, for consistency with other occurrences of $\mathsf{f}\ \mathsf{a}\ \mathsf{b}\ \mathsf{c}$.

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 the `binding` pointer is updated. To detect this situation, we store x's `binding` value in the $@(x, \overline{s})$ cell's `binding` field (which is otherwise unused). To find out whether the cache is valid, it suffices to check that the `binding` fields of x and $@(x, \overline{s})$ are equal.

Term Orders. Superposition provers rely on term orders to prune the search space. To ensure completeness, the order must be a simplification order that can be extended to a simplification order that is total on variable-free terms. The Knuth–Bendix order (KBO) and the lexicographic path order (LPO) meet this

criterion. KBO is generally regarded as the more robust and efficient option for superposition. E implements both. In earlier work, Blanchette and colleagues have shown that only KBO can be generalized gracefully while preserving all the necessary properties for superposition [5]. For this reason, we focus on KBO.

E implements the linear-time algorithm for KBO described by Löchner [19], which relies on the tupling method to store intermediate results, avoiding repeated computations. It is straightforward to generalize the algorithm to compute the graceful λfHOL version of KBO [5]. The main difference is that when comparing two terms $\mathsf{f}\ \overline{s_m}$ and $\mathsf{f}\ \overline{t_n}$, because of partial application we may now have $m \neq n$; this required changing the implementation to perform a length-lexicographic comparison of the tuples $\overline{s_m}$ and $\overline{t_n}$.

4 Unification and Matching

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

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

We present our unification procedure as a transition system, generalizing Baader and Nipkow [3]. 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.

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

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

Clash	$\{\mathsf{f}\ \overline{s} \stackrel{?}{=} \mathsf{g}\ \overline{t}\} \uplus S \Longrightarrow \bot$ if $\mathsf{f} \neq \mathsf{g}$
ClashTypeX	$\{x\ \overline{s_m} \stackrel{?}{=} u\ \overline{t_m}\} \uplus S \Longrightarrow \bot$ if x and u have different types
ClashLenXF	$\{x\ \overline{s_m} \stackrel{?}{=} \mathsf{f}\ \overline{t_n}\} \uplus S \Longrightarrow \bot$ if $m > n$
OccursCheck	$\{x \stackrel{?}{=} t\} \uplus S \Longrightarrow \bot$ if $x \in \mathcal{V}ar(t)$ and $x \neq t$

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

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

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. Soundness and completeness proofs as well as the pseudocode for unification and matching algorithms are included in our report [32].

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. Using this technique, Ehoh is virtually as efficient as E on first-order terms.

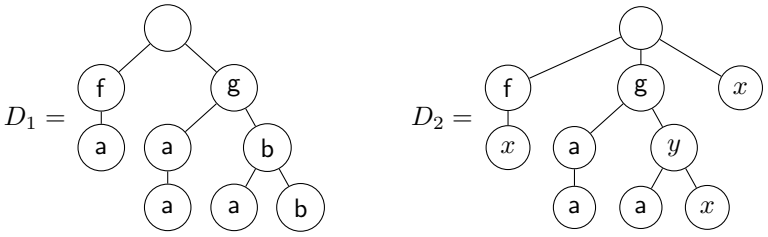
5 Indexing Data Structures

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 [20] and fingerprint indices [24], that needed to be generalized to λfHOL. It also uses feature vector indices [25] to speed up clause subsumption and related techniques, but these require no changes to work with λfHOL clauses.

Perfect Discrimination Trees. Discrimination trees [20] are tries in which every node is labeled with a symbol or a variable. A path from the root to a leaf node corresponds to a "serialized term"—a term expressed without parentheses and commas. Consider the following discrimination trees:



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

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

The data structure relies on the observation that serializing is unambiguous. Conveniently, this property also holds for λfHOL terms. Assume that two distinct λfHOL terms yield the same serialization. Clearly, they must disagree on parentheses; one will have the subterm $s\ t\ u$ where the other has $s\ (t\ u)$. However, these two subterms cannot both be well typed.

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

The main difficulty specific to λfHOL concerns applied variables. To enumerate all generalizing terms, E needs to backtrack from child to parent nodes. To achieve this, it relies on two stacks that store subterms of the query term: `term_stack` stores the terms that must be matched in turn against the current subtree, and `term_proc` stores, for each node from the root to the current subtree, the corresponding processed term, including any arguments yet to be matched.

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

A. If the node is labeled with a symbol f and the top item t of `term_stack` is $\mathsf{f}(\overline{t_n})$, replace t by n new items $t_1, \ldots, t_n$, and push t onto `term_proc`.
B. If the node is labeled with a variable x, there are two subcases. If x is already bound, check that $\sigma(x) = t$; otherwise, extend σ so that $\sigma(x) = t$. Next, pop a term t from `term_stack` and push it onto `term_proc`.

The goal is to reach an accepting node. If the query term and all the terms stored

in the tree are first-order, `term_stack` will then be empty, and the entire query term will have been matched.

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

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

	ϵ	g	g.b	g.b.a
`term_stack`:	[g(b, a)]	[b, a]	[a]	[]
`term_proc`:	[]	[g(b, a)]	[b, g(b, a)]	[a, b, g(b, a)]

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

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

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

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

Finally, to avoid traversing twice as many subterms as in the first-order case, we can optimize prefixes: Given a query term $\zeta\ \overline{t_n}$, we can also match prefixes $\zeta\ \overline{t_k}$, where $k < n$, by allowing `term_stack` to be nonempty at the end.

Fingerprint Indices. Fingerprint indices [24] 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. Given a term t and a position p, the *fingerprint function* $\mathcal{G}\mathit{fpf}$ is defined as

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

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

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

	f_1	f_2	A	B	N
f_1		✗	✗	✗	✗
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

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

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, possibly together with other terms that share the same fingerprint. This organization makes it possible to unify or match a query term s against all the terms T in one traversal. Once a node storing the terms $U \subseteq T$ has been reached, due to overapproximation we must apply unification or matching on s and each $u \in U$.

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

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

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

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

The most interesting new case is that of an applied variable. Given the sample positions $(\epsilon, 2, 1)$, the fingerprint of x is $(\mathsf{A}, \mathsf{B}, \mathsf{B})$ as before, whereas the fingerprint of $x\ \mathsf{c}$ is $(\mathsf{A}, \mathsf{B}, \mathsf{c})$. As another example, let $(\epsilon, 2, 1, 2.2, 2.1, 1.2, 1.1)$ be the sample

positions, and let $s = x\ (\mathsf{f}\ \mathsf{b}\ \mathsf{c})$ and $t = \mathsf{g}\ \mathsf{a}\ (y\ \mathsf{d})$. Their fingerprints are

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

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

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

6 Inference Rules

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

Ehoh implements essentially the same logical calculus as E, except that it is generalized to λfHOL terms. The standard inference rules and completeness proof of superposition can be reused verbatim; the only changes concern the basic definitions of terms and substitutions [7, Sect. 1].

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

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

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

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

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

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 over-approximate the set of all possible inference partners. The unification algorithm is then applied to filter out unsuitable candidates. Thanks to prefix optimization, we can avoid gracelessly polluting the index with all prefix subterms.

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

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:

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

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

E maintains a perfect discrimination tree that stores clauses of the form $s \approx t$ indexed by s and t. When applying the ES rule, E considers each 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 extension σ' of σ. If so, ES is applicable. To consider nonempty contexts, the prover traverses the subterms u' and v' of u and v in lockstep, as long as they appear under identical contexts. Thanks to prefix optimization, when Ehoh is given a subterm u', it can find an equation $s \approx t$ in the tree such that $\sigma(s)$ is equal to some prefix of u', with n arguments $\overline{u_n}$ remaining as unmatched. Checking for equality subsumption then amounts to checking that $v' = \sigma'(t)\ \overline{u_n}$, for some extension σ' of σ.

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

7 Heuristics

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

Term Order Generation. The inference rules and the redundancy criterion are parameterized by a term order (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 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. With the applicative encoding, generation schemes can take the symbols $@_{\tau,\upsilon}$ into account, effectively exploiting the type information carried by such symbols. To simulate this behavior, we introduced four generation schemes that extend E's existing symbol-frequency-based schemes by partitioning the symbols by type. To each symbol, the new schemes assign a frequency corresponding to the sum of all symbol frequencies for its class. In addition, we designed four schemes that combine E's type-agnostic and Ehoh's type-aware approaches.

To generate symbol precedences, E can sort symbols by weight and use the symbol's position in the sorted array as the basis for precedence. To account for the type information introduced by the applicative encoding, we implemented four type-aware precedence generation schemes.

Literal Selection. The side conditions of the superposition rules (SN and SP, Sect. 6) allow the use of a literal selection function to restrict the set of *inference literals*, thereby pruning the search space. Given a clause, a literal selection function returns a (possibly empty) subset of its literals. For completeness, any nonempty subset selected must contain at least one negative literal. If no literal is selected, all *maximal* literals become inference literals. The most widely used function in E is probably `SelectMaxLComplexAvoidPosPred`, which we abbreviate to `SelectMLCAPP`. It selects at most one negative literal, based on size, groundness, and maximality of the literal in the clause. It also avoids negative literals that share a predicate symbol with a positive literal in the same clause.

Clause Selection. Selection of the given clause is a critical choice point. E heuristically assigns *clause priorities* and *clause weights* to the candidates. E's main loop visits, in round-robin fashion, a set of priority queues. From each queue, it selects a number of clauses with the highest priorities, breaking ties by preferring smaller weights.

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`. It computes term and clause weights according to eight parameters, notably *conj_mul*, a multiplier applied to the weight of conjecture symbols. We implemented a new type-aware template function, called `ConjectureRelativeSymbolTypeWeight`, that applies the *conj_mul* multiplier to all symbols whose type occurs in the conjecture.

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, which analyzes the input problem and chooses a configuration known to perform well on similar problems. More recently, E has been extended with an *autoschedule* mode, which applies a portfolio of configurations in sequence on the given problem. Configurations that perform well on a wide range of problems have emerged over time. One of them is the configuration that is most often chosen by E's *auto* mode. We call it *boa* ("best of *auto*").

8 Evaluation

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

We set a CPU time limit of 60 s per problem. The experiments were performed on StarExec [28] nodes equipped with Intel Xeon E5-2609 0 CPUs clocked at 2.40 GHz and with 8192 MB of memory. Our raw data are publicly available.[2]

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*. All heuristic parameters were tested on a 5012 problem suite generated using Sledgehammer, consisting of four versions of the Judgment Day [11] suite. Our main findings are as follows:

- The combination of the weight generation scheme `invtypefreqrank` and the precedence generation scheme `invtypefreq` performs best.
- The literal selection heuristics `SelectMLCAPP`, `SelectMLCAPPPreferAppVar`, and `SelectMLCAPPAvoidAppVar` give virtually the same results.
- The clause selection function `ConjectureRelativeSymbolTypeWeight` with `ConstPrio` priority and an *appv_mul* factor of 1.41 performs best.

We derived a new configuration from *boa*, called *hoboa*, by enabling the features identified in the first and third points. Below, we present a more detailed evaluation of *hoboa*, along with other configurations, on a larger benchmark suite. The benchmarks are partitioned as follows: (1) 1147 first-order TPTP [29] problems belonging to the FOF (untyped) and TF0 (monomorphic) categories, excluding arithmetic; (2) 5012 Sledgehammer-generated problems from the Judgment Day [11] suite, targeting the monomorphic first-order logic embodied by TPTP TF0; (3) all 530 monomorphic higher-order problems from the TH0 category of the TPTP library belonging to the λfHOL fragment; (4) 5012 Judgment Day problems targeting the λfHOL fragment of TPTP TH0.

For the first group of benchmarks, we randomly chose 1000 FOF problems (out of 8172) and all monomorphic TFF problems that are parsable by E. 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 smaller and larger problems arising in interactive verification. Each subgroup itself consists of two sub-subgroups of 1253 problems, generated by using either λ-lifting or SK-style combinators to encode λ-expressions.

We evaluated Ehoh against Leo-III and Satallax and a version of E, called @+E, that first performs the applicative encoding. Leo-III and Satallax have

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

	First-order			Higher-order		
	TPTP	SH32	SH512	TPTP	SH32	SH512
E a	598	939	1234			
E as	**645**	950	**1311**			
E b	546	944	1243			
@+E a	526	943	1114	395	962	1119
@+E as	567	950	1151	397	965	1155
@+E b	538	942	1228	397	960	1272
Ehoh a	599	938	1233	396	962	1240
Ehoh as	644	949	1310	395	**973**	**1325**
Ehoh b	547	944	1243	396	966	1244
Ehoh hb	502	944	1231	393	968	1262
Leo-III	542	**951**	1126	**421**	963	1145
Satallax				406	768	790

Fig. 1. Number of proved problems

the advantage that they can instantiate higher-order variables by λ-terms. Thus, some formulas that are provable by these two systems may be nontheorems for @+E and Ehoh. A simple example is the conjecture $\exists f.\, \forall x\ y.\ f\ x\ y \approx \mathtt{g}\ y\ x$, whose proof requires taking $\lambda x\ y.\ \mathtt{g}\ y\ x$ as the witness for f.

We also evaluated E, @+E, Ehoh, and Leo-III on first-order benchmarks. The number of problems each system proved is given in Figure 1. We considered the E modes *auto* (a) and *autoschedule* (as) and the configurations *boa* (b) and *hoboa* (hb). We observe the following:

- Comparing the Ehoh rows with the corresponding E rows, we see that Ehoh's overhead is barely noticeable—the difference is at most one problem. The raw evaluation data reveal that Ehoh's time overhead is about 3.7%.
- Ehoh generally outperforms the applicative encoding, on both first-order and higher-order problems. On Sledgehammer benchmarks, the best Ehoh mode (*autoschedule*) clearly outperforms all @+E modes and configurations. Despite this, there are problems that @+E proves faster than Ehoh.
- Especially on large benchmarks, the E variants are substantially more successful than Leo-III and Satallax. On the other hand, Leo-III emerges as the winner on the first-order SH32 benchmark set, presumably thanks to the combination of first-order backends (CVC4, E, and iProver) it depends on.
- The new *hoboa* configuration outperforms *boa* on higher-order problems, suggesting that it could be worthwhile to re-train *auto* and *autoschedule* based on λfHOL benchmarks and to design further heuristics.

9 Discussion and Related Work

Most higher-order provers were developed from the ground up. Two exceptions are Otter-λ by Beeson [6] and Zipperposition by Cruanes [14]. Otter-λ adds λ-terms and second-order unification to the superposition-based Otter. The

approach is pragmatic, with little emphasis on completeness. Zipperposition is a superposition-based prover written in OCaml. It was initially designed for first-order logic but subsequently extended to higher-order logic. Its performance is a far cry from E's, but it is easier to modify. It is used by Bentkamp et al. [7] for experimenting with higher-order features. Finally, there is noteworthy preliminary work by the developers of Vampire [10] and of CVC4 and veriT [4].

Native higher-order reasoning was pioneered by Robinson [22], Andrews [1], and Huet [16]. TPS, by Andrews et al. [2], was based on expansion proofs and let users specify proof outlines. The Leo systems, developed by Benzmüller and his colleagues, are based on resolution and paramodulation. LEO [8] introduced the cooperative paradigm to integrate first-order provers. Leo-III [27] expands the cooperation with SMT (satisfiability modulo theories) solvers and introduces term orders. Brown's Satallax [12] is based on a higher-order tableau calculus, guided by a SAT solver; recent versions also cooperate with first-order provers.

An alternative to all of the above is to reduce higher-order logic to first-order logic by means of a translation. Robinson [23] outlined this approach decades before tools such as Sledgehammer [21] and HOLyHammer [17] popularized it in proof assistants. In addition to performing an applicative encoding, such translations must eliminate the λ-expressions and encode the type information.

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, encoding is incompatible with interpreted operators, notably for arithmetic. A further complication is that in a monomorphic logic, @ is not a single symbol but a type-indexed family of symbols $@_{\tau,\upsilon}$, which must be correctly introduced and recognized. Finally, the encoding must be undone in the generated 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.

10 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.

The resulting Ehoh prover will form the basis of our work towards strong higher-order automation. Our aim is to turn it into a prover that excels on proof obligations emerging from interactive verification; in our experience, these tend to be large but only mildly higher-order. Our next steps will be to extend E's term data structure with λ-expressions and investigate techniques for computing higher-order unifiers efficiently.

Acknowledgment. We are grateful to the maintainers of StarExec for letting us use their service. We thank Ahmed Bhayat, Alexander Bentkamp, Daniel El Ouraoui, Michael Färber, Pascal Fontaine, Predrag Janičić, Robert Lewis, Tomer Libal, Giles Reger, Hans-Jörg Schurr, Alexander Steen, Mark Summerfield, 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.

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., 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)
[3] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
[4] Barbosa, H., Reynolds, A., Fontaine, P., Ouraoui, D.E., Tinelli, C.: Higher-order SMT solving (work in progress). In: Dimitrova, R., D'Silva, V. (eds.) SMT 2018 (2018)
[5] Becker, H., Blanchette, J.C., Waldmann, U., Wand, D.: A transfinite Knuth–Bendix order for lambda-free higher-order terms. In: de Moura, L. (ed.) CADE-26. LNCS, vol. 10395, pp. 432–453. Springer (2017)
[6] Beeson, M.: Lambda logic. In: Basin, D.A., Rusinowitch, M. (eds.) IJCAR 2004. LNCS, vol. 3097, pp. 460–474. Springer (2004)
[7] Bentkamp, A., Blanchette, J.C., Cruanes, S., Waldmann, U.: Superposition for lambda-free higher-order logic. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS, vol. 10900, pp. 28–46. Springer (2018)
[8] Benzmüller, C., Kohlhase, M.: System description: LEO—a higher-order theorem prover. In: Kirchner, C., Kirchner, H. (eds.) CADE-15. LNCS, vol. 1421, pp. 139–144. Springer (1998)
[9] Benzmüller, C., Sultana, N., Paulson, L.C., Theiss, F.: The higher-order prover Leo-II. J. Autom. Reason. 55(4), 389–404 (2015)
[10] Bhayat, A., Reger, G.: Set of support for higher-order reasoning. In: Konev, B., Urban, J., Rümmer, P. (eds.) PAAR-2018. CEUR Workshop Proceedings, vol. 2162, pp. 2–16. CEUR-WS.org (2018)
[11] Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer (2010)
[12] Brown, C.E.: Satallax: An automatic higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 111–117. Springer (2012)
[13] Cruanes, S.: Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond. PhD thesis, École polytechnique (2015), `https://who.rocq.inria.fr/Simon.Cruanes/files/thesis.pdf`

[14] Cruanes, S.: Superposition with structural induction. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS, vol. 10483, pp. 172–188. Springer (2017)
[15] Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer (2013)
[16] Huet, G.P.: A mechanization of type theory. In: Nilsson, N.J. (ed.) IJCAI-73. pp. 139–146. William Kaufmann (1973)
[17] Kaliszyk, C., Urban, J.: HOL(y)Hammer: Online ATP service for HOL Light. Math. Comput. Sci. 9(1), 5–22 (2015)
[18] Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer (2013)
[19] Löchner, B.: Things to know when implementing KBO. J. Autom. Reason. 36(4), 289–310 (2006)
[20] McCune, W.: Experiments with discrimination-tree indexing and path indexing for term retrieval. J. Autom. Reason. 9(2), 147–167 (1992)
[21] Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Schulz, S., Ternovska, E. (eds.) IWIL-2010. EPiC, vol. 2, pp. 1–11. EasyChair (2012)
[22] Robinson, J.: Mechanizing higher order logic. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 151–170. Edinburgh University Press (1969)
[23] Robinson, J.: A note on mechanizing higher order logic. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 5, pp. 121–135. Edinburgh University Press (1970)
[24] Schulz, S.: Fingerprint indexing for paramodulation and rewriting. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 477–483. Springer (2012)
[25] Schulz, S.: Simple and efficient clause subsumption with feature vector indexing. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics—Essays in Memory of William W. McCune. LNCS, vol. 7788, pp. 45–67. Springer (2013)
[26] Schulz, S.: System description: E 1.8. In: McMillan, K.L., Middeldorp, A., Voronkov, A. (eds.) LPAR-19. LNCS, vol. 8312, pp. 735–743. Springer (2013)
[27] Steen, A., Benzmüller, C.: The higher-order prover Leo-III. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS, vol. 10900, pp. 108–116. Springer (2018)
[28] Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 367–373. Springer (2014)
[29] 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)
[30] Sutcliffe, G.: The CADE-26 automated theorem proving system competition—CASC-26. AI Commun. 30(6), 419–432 (2017)
[31] Vukmirović, P.: Implementation of Lambda-Free Higher-Order Superposition. MSc thesis, Vrije Universiteit Amsterdam (2018), `http://matryoshka.gforge.inria.fr/pubs/vukmirovic_msc_thesis.pdf`
[32] Vukmirović, P., Blanchette, J.C., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic (technical report). Technical report (2019), `http://matryoshka.gforge.inria.fr/pubs/ehoh_report.pdf`
[33] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 140–145. Springer (2009)