Extending a High-Performance Prover to Higher-Order Logic

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

[1] Vrije Universiteit Amsterdam, Amsterdam, the Netherlands
`{petar.vukmirovic2@gmail.com,j.c.blanchette@vu.nl}`
[2] Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
[3] DHBW Stuttgart, Stuttgart, Germany
`stephan.schulz@dhbw-stuttgart.de`

Abstract. Most users of proof assistants want more proof automation. Some proof assistants discharge goals by translating them to first-order logic and invoking an efficient prover on them, but much is lost in translation. Instead, we propose to extend first-order provers with native support for higher-order features. Building on our extension of E to λ-free higher-order logic, we extend E to full higher-order logic. The result is the strongest prover on benchmarks exported from a proof assistant.

1 Introduction

In the last few decades, proof assistants have become indispensable tools for developing trustworthy formal proofs. They are used both in academia to verify mathematical theories [14] and in industry to verify the correctness of hardware [18] and software [13, 19, 21]. However, due to the lack of strong built-in proof automation, proving seemingly simple goals can be a tedious manual task. To mitigate this, many proof assistants include a subsystem such as CoqHammer, HOL(y)Hammer, or Sledgehammer [7] that translates higher-order goals to first-order logic and passes them to efficient first-order automatic provers. If a first-order prover succeeds, the proof is reconstructed and the goal is closed.

Unfortunately, the translation of higher-order constructs is clumsy and leads to poor performance on goals that require higher-order reasoning. Using native higher-order provers such as Satallax [8] as backends is not always a good solution because they are much less efficient than their first-order counterparts [32]. To bridge this gap, in 2016 we proposed to develop a new generation of higher-order provers that extend the arguably most successful first-order calculus, superposition, to higher-order logic, starting from a position of strength.

Our research has focused on three milestones: supporting λ-free higher-order logic, adding λ-terms, and adding first-class Boolean terms. In 2019, we extended the state-of-the-art first-order prover E [29] with a λ-free superposition calculus [37], obtaining a version of E called Ehoh, as a stepping stone towards full higher-order logic. Together with Bentkamp, Tourret, and Waldmann, we have since developed calculi, called λ-*superposition*, corresponding to the other two milestones [2, 3] and implemented them in the experimental superposition prover

Zipperposition [11]. This OCaml prover is not nearly as efficient as E. Nevertheless, it has won the higher-order division of the CASC prover competition [34] in 2020, 2021, and 2022, ending nearly a decade of Satallax domination.

In this paper, we fulfill a three-year-old promise: We present the extension of Ehoh to full higher-order logic (Sect. 2) based on incomplete variants of λ-superposition. We call this prover λE. In λE's implementation, we used the extensive experience with Zipperposition to choose a set of effective rules that could easily be retrofitted into an originally first-order prover. Another principle that guided the design of λE was *gracefulness*: We made sure that our changes do not impact the strong first-order performance of E and Ehoh.

One of the main challenges we faced was retrofitting λ-terms in Ehoh's term representation (Sect. 3). Furthermore, Ehoh's inference engine assumes that inferences compute a most general unifier. We implemented a higher-order unification procedure [36] that can return multiple unifiers (Sect. 4) and integrated it in the inference engine. Finally, we extended and adapted the superposition rule, resulting in an incomplete, pragmatic variant of λ-superposition (Sect. 5).

We evaluated λE on a selection of proof assistants benchmarks as well as all higher-order theorems in the TPTP library [33] (Sect. 6). λE outperformed all other higher-order provers on the proof assistant benchmarks; on the TPTP benchmarks, it ended up second only to the cooperative version of Zipperposition, which employs Ehoh as a backend. An arguably fairer comparison without the backend puts λE in first place for both benchmark suites. We also compared the performance of λE with E on first-order problems and found that no overhead has been introduced by the extension to higher-order logic.

λE is part of the E prover's development repository and will be part of E 3.0. It can be enabled by passing the option `--enable-ho` to the `configure` script. E and λE's source code is freely available online.[1]

2 Logic

Our target logic is monomorphic classical higher-order logic with Hilbert choice. The following text is partly based on Vukmirović et al. [35, Sect. 2].

Terms s, t, u, v are inductively defined as free variables $F, X, \ldots$, bound variables $x, y, z, \ldots$, constants $\mathsf{f}, \mathsf{g}, \mathsf{a}, \mathsf{b}, \ldots$, applications $s\,t$, and λ-abstractions $\lambda x.\, s$. Bound variables may be *loose* (e.g., y in $\lambda x.\, y\,\mathsf{a}$) [24].

We let $s\,\bar{t}_n$ stand for $s\,t_1\,\ldots\,t_n$ and $\lambda\overline{x}_n.\, s$ for $\lambda x_1.\ldots\lambda x_n.\, s$. Every β-normal term can be written as $\lambda\overline{x}_m.\, s\,\bar{t}_n$, where s is not an application; we call s the *head* of the term. If s is a free variable, we call the term *flex*; otherwise, the term is *rigid*. A term of type o, where o is the distinguished Boolean type, is called a *formula*. A type whose type is of the form $\tau_1 \to \cdots \to \tau_n \to o$ is called a *predicate*. Logical symbols are part of the signature and may thus occur within terms. We write them in bold: $\boldsymbol{\neg}, \boldsymbol{\wedge}, \boldsymbol{\vee}, \boldsymbol{\rightarrow}, \boldsymbol{\leftrightarrow}, \boldsymbol{\approx}, \ldots$.

On top of the terms, we define some clausal structure. This structure is needed by λ-superposition. A literal l is an equation $s \approx t$ or a disequation $s \not\approx t$. A clause

[1] `https://github.com/eprover/eprover.git`

is a finite multiset of literals, interpreted and written disjunctively: $l_1 \vee \cdots \vee l_n$. Notice that the clause-level operators are not set in bold. Predicate literals are encoded as (dis)equations with T based on their sign; for example, $\mathsf{even}(x)$ is encoded as $\mathsf{even}(x) \approx \mathsf{T}$, and $\neg\,\mathsf{even}(x)$ as $\mathsf{even}(x) \not\approx \mathsf{T}$.

3 Terms

E is designed around perfect term sharing [22], a principle that we kept in Ehoh and λE: Any two structurally identical terms are guaranteed to be the same object in memory. This is achieved through term *cells*, which represent individual terms. Each cell has (among other fields) (1) `f_code`, an integer corresponding to the symbol at the head of the term (negative if the head is a free variable, positive otherwise); (2) `num_args`, corresponding to the number of arguments applied to the head; and (3) `args`, an array of size `num_args` of pointers to argument terms. We use the notation $\mathsf{f}(s_1, \ldots, s_n)$ to denote a cell whose `f_code` corresponds to f, `num_args` equals n, and `args` points to the cells for $s_1, \ldots s_n$.

Ehoh represents λ-free higher-order terms using a flattened, spine notation. Thus, the terms f, $\mathsf{f\,a}$, and $\mathsf{f\,a\,b}$ are represented by the cells f, $\mathsf{f}(\mathsf{a})$, and $\mathsf{f}(\mathsf{a}, \mathsf{b})$. To ensure that free variables are perfectly shared, Ehoh treats applied free variables differently: Arguments are not applied directly to a free variable, but using an internal symbol @ of variable arity. For example, the term $X\,\mathsf{a}\,\mathsf{b}$ is represented by the cell $@(X, \mathsf{a}, \mathsf{b})$. This ensures that two different occurrences of the free variable X correspond to the same object, which makes substitutions more efficient [37].

Representation of λ-Terms. To support full higher-order logic, Ehoh's λ-free cell data structure must be extended to support the λ binder. We use the locally nameless representation [10]: De Bruijn indices represent (possibly loose) bound variables, whereas we keep the current representation for free variables.

Extending the term representation of Ehoh with a new term kind involves intricate manipulation of the cell data structure. De Bruijn indices must be represented like other cells with either a negative or a positive `f_code`, but the code must clearly identify that the cell is a De Bruijn index.

Other than possibly being instantiated during β-reduction, De Bruijn indices mostly behave as constants. Therefore, we choose to represent De Bruijn indices using positive `f_code`s: The De Bruijn index of value i will have i as the `f_code`. To ensure De Bruijn indices are not mistaken for function symbols, we use the `properties` bitfield of the cell, which holds precomputed properties of the cell. We introduce the property `IsDBVar` to denote that the cell represents a De Bruijn index. De Bruijn indices are systematically created through a dedicated function that sets the `IsDBVar` property. When given the same De Bruijn index and type, this function always returns the same object. Finally, we guard all the functions and macros that manipulate function codes to check if the property `IsDBVar` is set. To ensure perfect sharing of De Bruijn indices, arguments to De Bruijn indices are applied like for free variables, using @.

Extending cells to support λ-abstraction is easier. Each λ-abstraction has the distinguished function code `LAM` as the head symbol and two arguments: (1) a

De Bruijn index 0 of the type of the abstracted variable; (2) the body of the λ-abstraction. Consider the term $\lambda x.\, \lambda y.\, \mathsf{f}\, x\, x$, where both x and y have the type ι. This term is represented as $\lambda\, \lambda\, \mathsf{f}\, \mathbf{1}\, \mathbf{1}$ in locally nameless representation, where bold numbers represent De Bruijn indices. In λE, the same term is represented by the cell $\mathtt{LAM}(\mathbf{0}, \mathtt{LAM}(\mathbf{0}, \mathsf{f}(\mathbf{1}, \mathbf{1})))$, where all De Bruijn variables have type ι.

The first argument of `LAM` is redundant, since it can be deduced from the type of the λ-abstraction. However, basic λ-term manipulation operations often require access to this term. We store it explicitly to avoid creating it repeatedly.

Efficient $\boldsymbol{\beta}$-Reduction. Terms are stored in $\beta\eta$-reduced form. As these two reductions are performed very often, they ought to be efficient. Ehoh performs β-reduction by reducing the leftmost outermost β-redex first. To represent β-redexes, E uses the `@` symbol. Thus, the term $(\lambda x.\, \lambda y.\, (x\, y))\, \mathsf{f}\, \mathsf{a}$ is represented by $\mathtt{@}(\mathtt{LAM}(\mathbf{0}, \mathtt{LAM}(\mathbf{0}, \mathtt{@}(\mathbf{1}, \mathbf{0}))), \mathsf{f}, \mathsf{a})$. Another option would have been to add arguments applied to λ-terms directly to the λ representation (as in $\mathtt{LAM}(\mathbf{0}, \mathtt{LAM}(\mathbf{0}, \mathtt{@}(\mathbf{1}, \mathbf{0})), \mathsf{f}, \mathsf{a}))$, but this would break the invariant that `LAM` has two arguments. Furthermore, replacing free variables with λ-abstractions (e.g., replacing X with $\lambda x.\, x$ in $\mathtt{@}(X, \mathsf{a})$) would require additional normalization.

A term can be β-reduced as follows: When a cell $\mathtt{@}(\mathtt{LAM}(\mathbf{0}, s), t)$ is encountered, the field `binding` (normally used to record the substitution for a free variable) of the cell $\mathbf{0}$ is set to t. Then s is traversed to instantiate every loose occurrence of $\mathbf{0}$ in s with `binding`, whose loose De Bruijn indices are shifted by the number of λ binders above the occurrence of $\mathbf{0}$ in s [17]. Next, this procedure is applied to the resulting term and its subterms, in leftmost outermost fashion.

λE's basic β-normalization works in this way, but it features a few optimizations. First, given a term of the form $(\lambda \overline{x}_n.\, s)\, \overline{t}_n$, λE replaces the bound variables x_i with t_i in parallel. By avoiding the construction of intermediate terms, this reduces the number of recursive function calls and calls to the cell allocator.

Second, in line with the gracefulness principle, we want λE to incur little (or no) overhead on first-order problems and to excel on higher-order problems with a large first-order component. If β-reduction is implemented naively, finding a β-redex involves traversing the entire term. On purely first-order terms, β-reduction is then a complete waste of time. To avoid this, we use Ehoh's perfectly shared terms and their `properties` field. We introduce the property `HasBetaReducibleSubterm`, which is set if a cell is β-reducible. Whenever a new cell that contains a β-reducible term as a direct subterm is shared, the property is set. Setting of the property is inductively continued when further superterms are shared. For example, in the term $t = \mathsf{f}\, \mathsf{a}\, (\mathsf{g}((\lambda x.\, x)\, \mathsf{a}))$, the cells for $(\lambda x.\, x)\, \mathsf{a}$, $\mathsf{g}\, ((\lambda x.\, x)\, \mathsf{a})$, and t itself have the property `HasBetaReducibleSubterm` set. When it needs to find β-reducible subterms, λE will visit only the cells with this property set. This further means that on first-order subterms, a single bit masking operation is enough to determine that no subterm should be visited.

Along similar lines, we introduce a property `HasDBSubterm` that caches whether the cell contains a De Bruijn subterm. This makes instantiating De Bruijn indices during β-normalization faster, since only the subterms that contain De Bruijn indices must be visited. Similarly, some other operations such as shifting

De Bruijn indices or determining whether a term is closed (i.e., it contains no loose bound variables) can be sped up or even avoided if the term is first-order.

Efficient η-Reduction. The term $\lambda x.\, s\, x$ is η-reduced to s whenever x does not occur unbound in s. We use the observation that a term cannot be η-reduced if it has no λ-abstraction subterms and introduce a property `HasLambda` that notes the presence of λ-abstraction in a term. Only terms with this property are visited during η-reduction.

λE performs parallel η-reduction: It recognizes terms of the form $\lambda \overline{x}_n.\, s\, \overline{x}_n$ such that none of the x_i occurs unbound in s. If done naively, reducing terms of this kind requires up to n traversals of s to check if each x_i occurs in s. In λE, exactly one traversal of s is required. More precisely, when η-reducing a cell $\mathtt{LAM}(\mathbf{0}, s)$, λE considers all λ binders in s as well. In general, the cell will be of the form $\mathtt{LAM}(\mathbf{0}, \ldots, \mathtt{LAM}(\mathbf{0}, t) \ldots)$, where t is not a λ-abstraction, and l is the number of `LAM` symbols above t. Then λE breaks the body t down into a maximal decomposition $u\, (\boldsymbol{n-1}) \;\ldots\; \mathbf{1}\, \mathbf{0}$. If $n = 0$, the cell is not η-reducible. Otherwise, u is traversed to determine the minimal index $\boldsymbol{j}$ of a loose De Bruijn index, taking $\boldsymbol{j} = \infty$ if no such index exists. λE can then remove the $k = \min\{j, l, n\}$ rightmost outermost λ binders in $\mathtt{LAM}(\mathbf{0}, \ldots, \mathtt{LAM}(\mathbf{0}, t) \ldots)$ and replace t by the variant of u $(\boldsymbol{n-1}) \;\ldots\; (\boldsymbol{k+1})\, \boldsymbol{k}$ obtained by shifting the loose De Bruijn indices down by k.

To illustrate this convoluted De Bruijn arithmetic, we consider the term $\lambda x.\, \lambda y.\, \lambda z.\, \mathsf{f}\, x\, x\, y\, z$. This term is represented by the cell $\mathtt{LAM}(\mathbf{0}, \mathtt{LAM}(\mathbf{0}, \mathtt{LAM}(\mathbf{0}, \mathsf{f}(\mathbf{2}, \mathbf{2}, \mathbf{1}, \mathbf{0}))))$. λE splits $\mathsf{f}(\mathbf{2}, \mathbf{2}, \mathbf{1}, \mathbf{0})$ into two parts: $u = \mathsf{f}\, \mathbf{2}$ and the arguments $\mathbf{2}, \mathbf{1}, \mathbf{0}$. Since the minimal index in u is $\mathbf{2}$, we can omit the De Bruijn indices $\mathbf{1}$ and $\mathbf{0}$ and their λ binders, yielding the η-reduced cell $\mathtt{LAM}(\mathbf{0}, \mathsf{f}(\mathbf{0}, \mathbf{0}))$.

Parallel η-reduction both speeds up η-reduction and avoids creating intermediate terms. For finding the minimal loose De Bruijn index, optimizations such as the `HasDBSubterm` property are used.

Representation of Boolean Terms. E and Ehoh represent Boolean terms using cells whose `f_code`s are reserved for logical symbols. Quantified formulas are represented by cells in which the first argument is the quantified variable and the second one is the body of the quantified formula. For example, the term $\forall x.\, \mathsf{p}\, x$ corresponds to the cell $\forall(X, \mathsf{p}(X))$, where X is a free variable. This representation is convenient for parsing and clausification, which is what E and Ehoh use it for, but in full higher-order logic, it is problematic during proof search: Booleans can occur as subterms in clauses, as in $\mathsf{q}(X) \lor \mathsf{p}(\forall(X, \mathsf{r}(X)))$, and instantiating X in the first literal should not affect X in the second literal.

To avoid this issue, in λE we use λ binders to represent quantified formulas. Thus, $\forall x.\, s$ is represented by $\forall\, (\lambda x.\, s)$. Quantifiers are then unary symbols that do not directly bind the variables. Since λE represents bound variables using De Bruijn indices, this solves the α-conversion issues. However, this solution is incompatible with thousands of decades-old lines of clausification code that assumes the E representation of quantified formulas. Therefore, λE converts quantified formulas only after clausification, for Boolean terms that occur in a higher-order context (e.g., as argument to a function symbol).

New Term Orders. The λ-superposition calculus is parameterized by a term order that is used to break symmetries in the search space. We implemented the versions of the Knuth–Bendix order (KBO) and lexicographic path order (LPO) for higher-order terms described by Bentkamp et al. [2]. These orders encode λ-terms as first-order terms and then invoke the standard KBO or LPO. For efficiency, we implemented separate KBO and LPO functions that compute the order directly, intertwining the encoding and the order computation.

Ehoh cells contain a `binding` field that can be used to store the substitution for a free variable. Substitutions can then be applied by following the `binding` pointers, replacing each free variable with its instance. Thus, when Ehoh needs to perform a KBO or LPO comparison of an instantiated term, it needs only follow the `binding` pointers. In full higher-order logic, however, instantiating a variable can trigger a chain of $\beta\eta$-reductions, changing the shape of the term dramatically. To prevent this, λE computes the $\beta\eta$-reduced instances of the terms before comparing them using KBO or LPO.

4 Unification, Matching, and Term Indexing

Standard superposition crucially depends on the concept of a most general unifier (MGU). In higher-order logic, such a unifier does not always exist, and the concept is replaced by that of a complete set of unifiers (CSU), which may be infinite. Vukmirović et al. [36] designed an efficient procedure to enumerate a CSU for a term pair. It is implemented in Zipperposition, together with some extensions to term indexing. In λE, we further improve the performance of this procedure by implementing a terminating, incomplete variant. We also introduce a new indexing data structure.

The Unification Procedure. The unification procedure works by maintaining a list of unification pairs to be solved. After choosing a pair, it first normalizes it by β-reducing and instantiating the heads of both terms in the pair. Then, if either head is a variable, it computes an appropriate binding for this variable, thereby approximating the solution.

Unlike in first-order and λ-free higher-order unification, in the full higher-order case there may be many bindings that lead to a solution. To reduce this mostly blind guessing of bindings, the procedure features support for *oracles* [36]. These are procedures that solve the unification problem for a subclass of higher-order terms on which unification is decidable and, for λE, unary. Oracles help increase performance, avoid nontermination, and avoid redundant bindings.

Vukmirović et al. described their procedure as a transition system. In λE, the procedure is implemented nonrecursively, and the unifiers are enumerated using an iterator object that encapsulates the state of the unifier search. The iterator consists of five fields: (1) *constraints*, which holds the unification constraints; (2) *bt_state*, a stack that contains information necessary to backtrack to a previous state; (3) *branch_iter*, which stores how far we are in exploring different possibilities from the current search node; (4) *steps*, which remembers how many different unification bindings (such as imitation, projection, and identification) are applied; and (5) *subst*, a stack storing the variables bound so far.

The iterator is initialized to hold the original problem in *constraints*, and all other fields are initially empty. The unifiers are retrieved one by one by calling the function ForwardIter. It returns True if the iterator made progress, in which case the unifier can be read via the iterator's *subst* field. Otherwise, no more unifiers can be found, and the iterator is no longer valid. The function's pseudocode is given below, including two auxiliary functions:

```
function NormalizeHead($t$) is
  if $t.head = @ \land t.args[0].is\_lambda()$ then
    reduce the top-level $\beta$-redex in $t$
    return NormalizeHead($t$)
  else if $t.head.is\_var() \land t.head.binding \neq$ Nil then
    $t.head \leftarrow t.head.binding$
    return NormalizeHead($t$)
  else
    return $t$

function BacktrackIter($iter$) is
  if $iter.bt\_state.empty()$ then
    clear all fields in $iter$
    return False
  else
    pop $(constraints, branch\_iter, steps, subst)$ from $iter.bt\_state$
    set the corresponding fields of $iter$
    return True

function ForwardIter($iter$) is
  $forward \leftarrow \neg\, iter.constraints.empty() \lor$ BacktrackIter($iter$)
  while $forward \land \neg\, iter.constraints.empty()$ do
    $(lhs, rhs) \leftarrow$ pop pair from $iter.constraints$
    $lhs \leftarrow$ NormalizeHead($lhs$)
    $rhs \leftarrow$ NormalizeHead($rhs$)
    normalize and discard the $\lambda$ prefixes of $lhs$ and $rhs$
  if $\neg lhs.head.is\_var() \land rhs.head.is\_var()$ then
    swap $lhs$ and $rhs$
  if $lhs.head.is\_var()$ then
    $oracle\_res \leftarrow$ Fixpoint($lhs, rhs, iter.subst$)
  if $oracle\_res =$ NotInFragment then
    $oracle\_res \leftarrow$ Pattern($lhs, rhs, iter.subst$)
    if $oracle\_res =$ NotUnifiable then
      $forward \leftarrow$ BacktrackIter(iter)
    else if $oracle\_res =$ NotInFragment then
      $n\_steps, n\_branch\_iter, n\_binding \leftarrow$
        NextBinding($lhs, rhs, iter.steps, iter.branch\_iter$)
      if $n\_branch\_iter \neq$ BindEnd then
        push pair (lhs,rhs) back to $iter.constraints$
        push quadruple ($iter.constraints$, $n\_branch\_iter$,
            $iter.steps$, $iter.subst$) onto $iter.bt\_state$
        extend $iter.subst$ with $n\_binding$
        $iter.steps \leftarrow n\_steps$
        $iter.branch\_iter \leftarrow$ BindBegin
      else if $lhs.head = rhs.head$ then
        create constraint pairs of arguments of $lhs$ and $rhs$
            and push them to $iter.constraints$
        $iter.branch\_iter \leftarrow$ BindBegin
  else if $lhs.head = rhs.head$ then
    create constraint pairs of arguments of $lhs$ and $rhs$
        and push them to $iter.constraints$
  else
    $forward \leftarrow$ BacktrackIter($iter$)
  return $forward$
```

ForwardIter begins by backtracking if the previous attempt was successful (i.e., all constraints were solved). If it finds a state from which it can continue, it takes term pairs from *constraints* until there are no more constraints or it is determined that no unifier exists. The terms are normalized by instantiating the head variable with its binding and reducing the potential top-level β-redex that might appear. This instantiation and reduction process is repeated until there are no more top-level β-redexes and the head is not a variable bound to some term. Then the term with shorter λ prefix is expanded (only on the top level) so that both λ prefixes have the same length. Finally, the λ prefix is ignored, and we focus only on the body. In this way, we avoid fully substituting and normalizing terms and perform just enough operations to determine the next step of the procedure.

If either term of the constraint is flex, we first invoke oracles to solve the constraint. λE implements the most efficient oracles implemented in Zipperposition: fixpoint and pattern [36, Sect. 6]. An oracle can return three results: (1) there is an MGU for the pair (Unifiable), which is recorded in *subst*, and the next pair in *constraints* is tried; (2) no MGU exists for the pair (NotUnifiable), which causes the iterator to backtrack; (3) if the pairs do not belong to the subclass that oracle can solve (NotInFragment), we generate possible variable bindings—that is, we guess the approximate form of the solution.

λE has a dedicated module that generates bindings (NextBinding). This module is given the current constraint and the values of *branch_iter* and *steps*, and it either returns the next binding and the new values of *branch_iter* and *steps* or reports that all different variable bindings are exhausted. The bindings that λE's unification procedure creates are imitation, Huet-style projection, identification, and elimination (one argument at a time) [36, Sect. 3]. A limit on the total number of applied binding rules can be set, as well as a limit on the number of individual rule applications. The binding module checks whether limits are reached using the iterator's *steps* field.

Computing bindings is the only point in the procedure where the search tree branches and different possibilities are explored. Thus, when λE follows the branch indicated by the binding module, it records the state to which it needs to return should the followed branch be backtracked. The state consists of the values of *constraints*, *steps*, and *subst* before the branch is followed and the value of *branch_iter* that points past the followed branch. The values of *branch_iter* are either BindBegin, which denotes that no binding was created, intermediate values that NextBinding uses to remember how far through bindings it is, and BindEnd, which indicates that all bindings are exhausted.

If all bindings are exhausted, the procedure checks whether the pair is flex–flex and both sides have the same head. If so, the pair is decomposed and constraints are derived from the pair's arguments; otherwise, the iterator backtracks. If the pair is rigid–rigid, for unification to succeed, the heads of both sides must be the same. Unification then continues with new constraints derived from the arguments. Otherwise, the iterator must be backtracked.

Matching. In E, the matching algorithm is mostly used inside simplification rules such as demodulation and subsumption [26]. As these rules must be efficiently performed, using a complex matching algorithm is not viable. Instead, we provide a matching algorithm for the pattern class of terms [24] to complement Ehoh's λ-free higher-order matching algorithm [37, Sect. 4]. A term is a *pattern* if each of its free variables either has no arguments (as in first-order logic) or is applied to distinct De Bruijn indices.

To determine which of the two algorithms to call (pattern or λ-free), we introduce a cached property `HasNonPatternVar`, which is set for terms of the form $X\,\overline{s}_n$ where $n > 0$ and either there exists some s_i that is not a De Bruijn index or there exist indices $i < j$ such that $s_i = s_j$ is a De Bruijn index. This property is propagated to the superterms when they are perfectly shared. This allows later checks if a term belongs to the pattern class to be performed in constant time.

We modify the λ-free higher-order matching algorithm to treat λ prefixes as above in the unification procedure—by bringing the prefixes to the same length and ignoring them afterwards. This ensures that the algorithm will never try to match a free variable with a λ-abstraction, making sure that β-redexes never appear. We also modify the algorithm to ensure that free variables are never bound to terms that have loose bound variables. This algorithm cannot find many complex matching substitutions (matchers), but it can efficiently determine whether two terms are variable renamings of each other or whether a simple matcher can be used, as in the case of $(X\ (\lambda x.\,x)\ \mathsf{b}, \mathsf{f}\ (\lambda x.\,x)\ \mathsf{b})$, where $X \mapsto \mathsf{f}$ is usually the desired matcher. If this algorithm does not find a matcher and both terms are patterns, pattern matching is tried.

Indexing. E, like other modern theorem provers, efficiently retrieves unifiable or matchable pairs of terms using indexing data structures. To find terms unifiable with a query term or instances of a query term, it uses *fingerprint indexing* [27]. Vukmirović et al. extended this data structure to support full higher-order terms

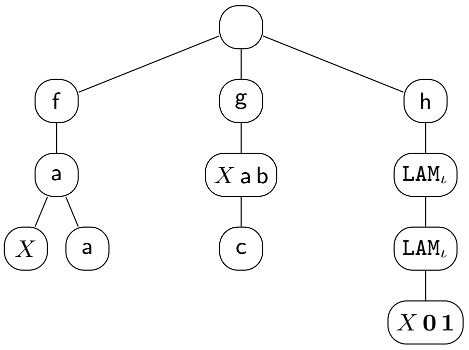

Fig. 1. First-order, λ-free higher-order, and higher-order pattern terms in a perfect discrimination tree

in Zipperposition [36, Sect. 6]. We use the same approach in λE, and we extend feature vector indices [28] in the same way.

E uses *perfect discrimination trees* [23] to find generalizations of the query term (i.e., terms of which the query term is an instance). This data structure is a trie that indexes terms by representing them in a serialized, flattened form. The left branch from the root in Figure 1 shows how the first-order terms $\mathsf{f}\,\mathsf{a}\,X$ and $\mathsf{f}\,\mathsf{a}\,\mathsf{a}$ are stored. In Ehoh, this data structure is extended to support partial application and applied variables [37].

In λE, we extend this structure to support λ-abstractions and the higher-order pattern matching algorithm. To this end, we change the way in which terms are serialized. First, we require that all terms are fully η-expanded (except for arguments of variables applied in patterns). Then, when the term is serialized, we use a single node for applied variable terms $X\,\overline{s}_n$, instead of a node for X followed by nodes for the arguments $\overline{s}_n$. We serialize λ-abstraction $\lambda x.\, s$ using a dedicated node $\mathtt{LAM}_\tau$, where τ is the type of x, followed by the serialization of s. Other than these changes, serialization remains as in Ehoh, following the gracefulness principle. Figure 1 shows how $\mathsf{g}\,(X\,\mathsf{a}\,\mathsf{b})\,\mathsf{c}$ and $\mathsf{h}\,(\lambda x.\,\lambda y.\,X\,y\,x)$ are serialized. Since the terms are stored in serialized form, it is hard to manipulate λ prefixes of stored terms during matching. Performing η-expansion when serializing terms ensures that matchable terms have λ prefixes of the same length.

We have dedicated separate nodes for applied variables because access to arguments of applied variables is necessary for the pattern matching algorithm. Even though arguments can be obtained by querying the arity n of the variable and taking the next n arguments in the serialization, this is both inefficient and inelegant. As for De Bruijn indices, we treat them the same as function symbols.

Following the notation from the extension of perfect discrimination trees to λ-free higher-order logic [37], we now describe how enumeration of generalizations is performed. To traverse the tree, λE begins at the root node and maintains two stacks: `term_stack` and `term_proc`, where `term_stack` contains the subterms of the query term that have to be matched, and `term_proc` contains processed terms that are used to backtrack to previous states. Initially, `term_stack` contains the query term, the current matching substitution σ is empty, and the successor node is chosen among the child nodes as follows:

A. If the node is labeled with a symbol ξ (where ξ is either a De Bruijn index or a constant) and the top item t of `term_stack` is of the form $\xi\,\overline{t}_n$, replace t by n new items $t_1, \dots, t_n$, and push t onto `term_proc`.
B. If the node is labeled with a symbol `LAM`$_\tau$ and the top item t of `term_stack` is of the form $\lambda x.\, s$ and the type of x is τ, replace t by s, and push t onto `term_proc`.
C. If the node is labeled with a possibly applied variable $X\,\overline{s}_n$ (where $n \geq 0$), and the top item of `term_stack` is t, the matching algorithm described above is run on $X\,\overline{s}_n$ and t. The algorithm takes into account σ built so far and extends it if necessary. If the algorithm succeeds, pop t from `term_stack`, push it onto `term_proc`, and save the original value of σ in the node.

Backtracking works in the opposite direction: If the current node is labeled with a De Bruijn index or function symbol node of arity n, pop n terms from `term_stack` and move the top of `term_proc` to `term_stack`. If the node is labeled with `LAM`$_\tau$, pop the top of `term_stack` and move the top of `term_proc` to `term_stack`. Finally, if the node is labeled with a possibly applied variable, move the top of the `term_proc` to `term_stack` and restore the value of σ.

As an example of how finding a generalization works, consider the following states of stacks and substitutions, which emerge when looking for generalizations of g (f a b) c in the tree of Figure 1:

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

5 Preprocessing, Calculus, and Extensions

Ehoh's simple λ-free higher-order calculus performed well on Sledgehammer problems and formed a promising stepping stone to full higher-order logic [37]. When implementing support for full higher-order logic, we were guided by efficiency and gracefulness with respect to Ehoh's calculus rather than completeness. Whereas Zipperposition provides both complete and incomplete modes, λE only offers incomplete modes.

Preprocessing. Our experience with Zipperposition showed the importance of flexibility in preprocessing the higher-order problems [35]. Therefore, we implemented a flexible preprocessing module in λE.

To maintain compatibility with Ehoh, λE can optionally transform all λ-abstractions into named functions. This process is called λ-*lifting* [16]. λE also removes all occurrences of Boolean subterms (other than $\top, \bot$, and free variables) in higher-order contexts using a FOOL-like transformation [20]. For example, the formula $\mathsf{f}(\mathsf{p} \wedge \mathsf{q}) \approx \mathsf{a}$ becomes $(\mathsf{p} \wedge \mathsf{q} \rightarrow \mathsf{f}(\top) \approx \mathsf{a}) \wedge (\neg\,(\mathsf{p} \wedge \mathsf{q}) \rightarrow \mathsf{f}(\bot) \approx \mathsf{a})$.

Many TPTP problems use the `definition` role to identify the definitions of symbols. λE can treat definition axioms as rewrite rules, and replace all occurrences of defined symbols during preprocessing. Furthermore, during SInE [15] axiom selection, it can always include the defined symbol in the trigger relation.

Calculus. λE implements the same superposition calculus as Ehoh with three important changes. First, wherever Ehoh requires the MGU of terms, λE enumerates unifiers from a finite subset of the CSU, as explained in Sect. 4. Second, λE uses versions of the KBO and LPO orders designed for λ-terms.

The third difference is more subtle. One of the main features of Ehoh is *prefix optimization* [37, Sect. 1]: a method that, given a demodulator $s \approx t$, makes it possible to replace both applied and unapplied occurrences of s by t by traversing only the first-order subterms of a rewritable term. In a λ-free setting, this optimization is useful, but in the presence of $\beta\eta$-normalization, the shapes of terms can change drastically, making it much harder to track prefixes of terms. This is why we disable the prefix optimization in λE. To compensate for losing this optimization, we introduce the argument congruence rule AC in λE and enable positive and negative functional extensionality (PE and NE) by default:

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

AC and NE assume that s and t are of function type. In NE, $\overline{X}$ denotes all the free variables occurring in s and t, and sk is a fresh Skolem symbol of the appropriate type. PE has a side condition that X may not occur in s, t, or C.

Saturation. E's saturation procedure assumes that each attempt to perform an inference will either result in a single clause or fail due to one of the inference side conditions. Unification procedures that produce multiple substitutions break this invariant, and the saturation procedure needed to be adjusted.

For Zipperposition, Vukmirović et al. developed a variant of the saturation procedure that interleaves computing unifiers and scheduling inferences to be performed [35]. Since completeness was not a design goal for λE, we did not implement this version of the saturation procedure. Instead, in places where previously a single unifier was expected, λE consumes all elements of the iterator used for enumerating a unifier, converting them into clauses.

Reasoning about Formulas. Even though most of the Boolean structure is removed during preprocessing, formulas can reappear at the top level of clauses during saturation. For example, after instantiating X with $\lambda x.\,\lambda y.\,x \wedge y$, the clause $X\,\mathsf{p}\,\mathsf{q} \vee \mathsf{a} \approx \mathsf{b}$ becomes $(\mathsf{p} \wedge \mathsf{q}) \vee \mathsf{a} \approx \mathsf{b}$. λE converts every clause of the form $\varphi \vee C$, where φ has a logic symbol as its head, or it is a (dis)equation between two formulas different than $\top$, to an explicitly quantified formula. Then, the clausification algorithm is invoked on the formula to restore the clausal structure. Zipperposition features more dynamic clausification modes, but for simplicity we decided not to implement them in λE.

The λ-superposition calculus for full higher-order logic [2] includes many rules that act on Boolean subterms, which are necessary for completeness. Other than Boolean simplification rules, which use simple tautologies such as $\mathsf{p} \land \top \leftrightarrow \mathsf{p}$ to simplify terms, we have implemented none of the Boolean rules of this calculus in λE. First, we have observed that complicated rules such as FluidBoolHoist and FluidLoobHoist are hardly ever useful in practice and usually only contribute to an uncontrolled increase in the proof state size. Second, simpler rules such as BoolHoist can usually be simulated by pragmatic rules that perform Boolean extensionality reasoning, described below.

To make up for excluding Boolean rules, we use an incomplete, but more easily controllable and intuitive rule, called *primitive instantiation*. This rule instantiates free predicate variables with approximations of formulas that are ground instances of this variable. We use the approximations described by Vukmirović and Nummelin [38, Sect. 3.3] and implemented them in a similar manner.

λE's handling of the Hilbert choice operator is inspired by Leo-III's [30]. λE recognizes clauses of the form $\neg\, P\, X \lor P\,(\mathsf{f}\, P)$, which essentially denote that f is a choice symbol. Then, when subterm $\mathsf{f}\, s$ is found during saturation, s is used to instantiate the choice axiom for f. Similarly, Leibniz equality [38] is eliminated by recognizing clauses of the form $\neg\, P\,\mathsf{a} \lor P\,\mathsf{b} \lor C$. These clauses are then instantiated with $P \mapsto \lambda x.\, x \approx \mathsf{a}$ and $P \mapsto \lambda x.\, x \not\approx \mathsf{b}$, which results in $\mathsf{a} \approx \mathsf{b} \lor C$.

Finally, λE treats induction axioms specially. Like Zipperposition [35, Sect. 4], it abstracts literals from the goal clauses and instantiates induction axioms with these abstractions. Since Zipperposition supports dynamic calculus-level clausification, induction axioms are instantiated during saturation, when the axioms are processed. In λE, this instantiation is performed immediately after clausification. After λE has collected all the abstractions, it traverses the clauses and instantiates those that have applied variable of the same type as the abstraction.

Extensionality. λE takes a pragmatic approach to reasoning about functional and Boolean extensionality: It uses *abstracting* rules [3] which simulate basic superposition calculus rules but do not require unifiability of the partner terms in the inference. More precisely, assume a core inference needs to be performed between two β-reduced terms u and v, such that they can be represented as $u = C[s_1, \ldots, s_n]$ and $v = C[t_1, \ldots, t_n]$, where C is the most general (green [3]) common context of u and v, not all of s_i and t_j are free variables, and for at least one i, $s_i \neq t_i$, s_i and t_i are not possibly applied free variables, and they are of Boolean or function type. Then, the conclusion is formed by taking the conclusion D of the core inference rule (which would be created if s and t are unifiable) and adding literals $s_1 \not\approx t_1 \lor \cdots \lor s_n \not\approx t_n$.

These rules are particularly useful because λE has no rules that dynamically process Booleans in FOOL-like fashion, such as BoolHoist. For example, given the clauses $\mathsf{f}\,(\mathsf{p} \land \mathsf{q}) \approx \mathsf{a}$ and $\mathsf{g}\,(\mathsf{f}\,\mathsf{p}) \not\approx \mathsf{b}$, the abstracting version of the superposition rule would result in $\mathsf{g\,a} \not\approx \mathsf{b} \vee (\mathsf{p} \wedge \mathsf{q}) \not\approx \mathsf{p}$. In this way, the Boolean structure bubbles up to the top level and is further processed by clausification. We noticed that this alleviates the need for the other Boolean rules in practice.

6 Evaluation

We now try to answer two questions about λE: *How does λE compare against other higher-order provers (including Ehoh)? Does λE introduce any overhead compared with Ehoh?* To answer these questions, we ran provers on problems from the TPTP library [33] and on benchmarks generated by Sledgehammer (SH) [25]. The experiments were carried out on StarExec Miami [31] nodes equipped with Intel Xeon E5-2620 v4 CPU clocked at 2.10 GHz. For the TPTP part, we used the CASC 2021[2] time limits: 120 s wall-clock and 960 s CPU. For SH benchmarks and to answer the other question, we used Sledgehammer's default time limit: 30 s wall-clock and CPU. The raw evaluation data is available online.[3]

Comparison with Other Provers. To answer the first question, we let λE compete with the top contenders in the higher-order division of CASC 2021: cvc5 0.0.7,[4] Ehoh 2.7 [37], Leo-III 1.6.6 [30], Vampire 4.6 [6], and Zipperposition 2.1 [35]. We also included Satallax 3.5 [8]. We used all 2899 higher-order theorems in TPTP 7.5.0 as well as 5000 SH higher-order benchmarks originating from the Seventeen benchmark suite [12]. On SH benchmarks, cvc5, Ehoh, λE, Vampire, and Zipperposition were run using custom schedules provided by their developers, optimized for single-core usage and low timeouts. Otherwise, we used the corresponding CASC configurations.

Although it focuses on λ-free higher-order logic, Ehoh 2.7 can parse full higher-order logic using λ-lifting. We included two versions of Zipperposition: *coop* uses Ehoh 2.7 as a backend to finish proof attempts, whereas *uncoop* does not use this feature. Both Ehoh and λE were run in the automatic scheduling mode. Compared to Ehoh, λE features a redesigned module for automatic scheduling, it can use multiple CPU cores, and its heuristics have been trained better on higher-order problems.

The results are shown in Figure 2. λE dramatically improves E's higher-order reasoning capabilities compared with Ehoh. It solves 20% more problems on TPTP benchmarks and 7% more problems on SH benchmarks, where Ehoh was already very successful. λE was mainly designed as an efficient backend to proof assistants. As such, it excels on SH benchmarks, outperforming the competition. On TPTP, it outperforms all higher-order provers other than Zipperposition-coop. If Zipperposition's Ehoh backend is disabled, λE outperforms Zipperposition by a wide margin. This comparison is arguably fairer; after all, λE does not use an older version of Zipperposition as a backend. These results suggest that λE

[2] `http://www.tptp.org/CASC/28/`
[3] `https://doi.org/10.5281/zenodo.6389849`
[4] `https://cvc5.github.io/`

	TPTP	SH
cvc5	1931	2577
Ehoh	2105	2611
λE	2533	**2804**
Leo-III	2282	1601
Satallax	2320	1719
Vampire	2203	2240
Zipperposition-coop	**2583**	2754
Zipperposition-uncoop	2483	2181

Fig. 2. Comparison of higher-order provers

	TPTP
Ehoh FO	535
Ehoh HO	538
λE FO	537
λE HO	**541**

Fig. 3. Evaluation of λE's overhead

already implements most of the necessary features for a high-performance higher-order prover but could benefit from the kind of fine-tuning that Zipperposition underwent in the last three years.

Remarkably, the raw evaluation data reveals thats λE solves 181 SH problems and 24 TPTP problems that Zipperposition-coop does not. The lower number of uniquely solved TPTP problems is likely because Zipperposition was heavily optimized on the TPTP.

Comparison with the First-Order E. Both Ehoh and λE can be compiled in a mode that disables most of the higher-order reasoning. This mode is designed for users that are interested only in E's first-order capabilities and care a lot about performance. To answer the second evaluation question, about assessing overhead of λE, we chose all the 1138 unique problems used at CASC from 2019 to 2021 in the first-order theorem division and ran Ehoh and λE both in this first-order (FO) mode and in higher-order (HO) mode.

We fixed a single configuration of options, because Ehoh's and λE's automatic scheduling methods could select different configurations and we would not be measuring the overhead but the quality of the chosen configurations. We chose the *boa* configuration [37, Sect. 7], which is the configuration most often used by E 2.2 in its automatic scheduling mode. The results are shown in Figure 3.

Counterintuitively, the higher-order versions of both provers outperform the first-order counterparts. However, the difference is so small that it can be attributed to the changes to memory layout that affect the order in which clauses are chosen. Similar effects are visible when comparing the first-order versions.

CASC Results. λE also took part in CASC 2022. In the TPTP higher-order division, λE finished second, after Zipperposition, as expected. In the Sledgehammer division, λE tied with Ehoh for first place, a disappointment. The likely explanation is that λE used a wrong configuration in this division, as we found out afterwards. We expect better performance at CASC 2023.

7 Discussion and Related Work

On the trajectory to λE, Bentkamp et al. developed three superposition calculi: for λ-free higher-order logic [4], for a higher-order logic with λ-abstraction but no Booleans [3], and for full higher-order logic [3]. These milestones allowed us to carefully estimate how the increased reasoning capabilities of each calculus influence its performance.

Extending first-order provers with higher-order reasoning capabilities has been attempted by other researchers as well. Barbosa et al. extended the SMT solvers CVC4 (now cvc5) and veriT to higher-order logic in an incomplete way [1]. Bhayat and Reger first extended Vampire to higher-order logic using combinatory unification [6], an incomplete approach, before they designed and implemented a complete higher-order superposition calculus based on SKBCI-combinators [5]. The advantage is that combinators can be supported as a thin layer on top of λ-free terms. This calculus is also implemented in Zipperposition. However, in informal experiments, we found that λ-superposition performs substantially better, corroborating the CASC results, so we decided to make a more profound change to Ehoh and implement λ-superposition.

Possibly the only actively maintained higher-order provers built from the bottom up as higher-order provers are Leo-III [30] and Satallax's [8] successor Lash [9]. A further overview of other traditional higher-order provers and the calculi they are based on can be found in the paper about Ehoh [37, Sect. 9].

8 Conclusion

In 2019, the reviewers of our Ehoh paper [37] were skeptical that extending Ehoh with support for full higher-order logic would be feasible. One of them wrote:

> A potential criticism could be that this step from E to Ehoh is just extending FOL by those aspects of HOL that are easily in reach with rather straightforward extensions (none of the extensions is indeed very complicated), and that the difficult challenges of fully supporting HOL have yet to be confronted.

We ended up addressing the theoretical "difficult challenges" in other work with colleagues. In this paper, we faced the practical challenges pertaining to the extension of Ehoh's data structures and algorithms to support full higher-order logic and demonstrated that such an extension is possible. Our evaluation shows that this extension makes λE the best higher-order prover on benchmarks coming from interactive theorem proving practice, which was our goal. λE lags slightly behind Zipperposition on TPTP problems. One reason might be that Zipperposition does not assume a clausal structure and can perform subtle formula-level inferences. It would be useful to implement the same features in λE. We have also only started tuning λE's heuristics on higher-order problems.

Acknowledgment. Ahmed Bhayat and Martin Suda provided us with Sledgehammer-optimized configurations for Vampire. Andrew Reynolds did the same for cvc5. Simon Cruanes, Wan Fokkink, and Mark Summerfield suggested several textual improvements. We thank them all.

This 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). Vukmirović and Blanchette have received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

References

[1] Barbosa, H., Reynolds, A., El Ouraoui, D., Tinelli, C., Barrett, C.W.: Extending SMT solvers to higher-order logic. In: CADE. LNCS, vol. 11716, pp. 35–54. Springer (2019)

[2] Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for full higher-order logic. In: Platzer, A., Sutcliffe, G. (eds.) CADE. LNCS, vol. 12699, pp. 396–412. Springer (2021)

[3] Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas. J. Autom. Reason. 65(7), 893–940 (2021)

[4] 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. LNCS, vol. 10900, pp. 28–46. Springer (2018)

[5] Bhayat, A., Reger, G.: Restricted combinatory unification. In: Fontaine, P. (ed.) CADE. LNCS, vol. 11716, pp. 74–93. Springer (2019)

[6] Bhayat, A., Reger, G.: A combinator-based superposition calculus for higher-order logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR (1). LNCS, vol. 12166, pp. 278–296. Springer (2020)

[7] Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formaliz. Reason. 9(1), 101–148 (2016)

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

[9] Brown, C.E., Kaliszyk, C.: Lash 1.0 (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. Lecture Notes in Computer Science, vol. 13385, pp. 350–358. Springer (2022)

[10] Charguéraud, A.: The locally nameless representation. J. Autom. Reason. 49(3), 363–408 (2012)

[11] Cruanes, S.: Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond. PhD thesis, École Polytechnique (2015)

[12] Desharnais, M., Vukmirović, P., Blanchette, J., Wenzel, M.: Seventeen provers under the hammer. In: Andronick, J., de Moura, L. (eds.) ITP. LIPIcs, vol. 237, pp. 8:1–8:18. Schloss Dagstuhl (2022)

[13] Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: Keeton, K., Roscoe, T. (eds.) OSDI. pp. 653–669. USENIX Association (2016)

[14] Hales, T.C., Adams, M., Bauer, G., Dang, D.T., Harrison, J., Hoang, T.L., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., Nguyen, T.Q., Nipkow, T., Obua, S., Pleso, J., Rute, J., Solovyev, A., Ta, A.H.T., Tran, T.N., Trieu, D.T., Urban, J., Vu, K.K., Zumkeller, R.: A formal proof of the Kepler conjecture. CoRR abs/1501.02155 (2015)

[15] Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE. LNCS, vol. 6803, pp. 299–314. Springer (2011)

[16] Hughes, R.J.M.: Super combinators: A new implementation method for applicative languages. In: Park, D.M.R., Friedman, D.P., Wise, D.S., Jr., G.L.S. (eds.) LFP. pp. 1–10. ACM (1982)

[17] Kamareddine, F.: Reviewing the classical and the de Bruijn notation for λ-calculus and pure type systems. J. Log. Comput. 11(3), 363–394 (2001)

[18] Kern, C., Greenstreet, M.R.: Formal verification in hardware design: A survey. ACM Trans. Design Autom. Electr. Syst. 4(2), 123–193 (1999)

[19] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating-system kernel. Commun. ACM 53(6), 107–115 (2010)

[20] Kotelnikov, E., Kovács, L., Suda, M., Voronkov, A.: A clausal normal form translation for FOOL. In: Benzmüller, C., Sutcliffe, G., Rojas, R. (eds.) GCAI. EPiC, vol. 41, pp. 53–71. EasyChair (2016)

[21] Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)

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

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

[24] Nipkow, T.: Functional unification of higher-order patterns. In: Best, E. (ed.) LICS. pp. 64–74. IEEE Computer Society (1993)

[25] 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. EPiC, vol. 2, pp. 1–11. EasyChair (2012)

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

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

[28] 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)

[29] Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) CADE. LNCS, vol. 11716, pp. 495–507. Springer (2019)

[30] Steen, A., Benzmüller, C.: Extensional higher-order paramodulation in Leo-III. J. Autom. Reason. 65(6), 775–807 (2021)

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

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

[33] 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)

[34] Sutcliffe, G.: The 10th IJCAR automated theorem proving system competition—CASC-J10. AI Commun. 34(2), 163–177 (2021)

[35] Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making higher-order superposition work. In: Platzer, A., Sutcliffe, G. (eds.) CADE. LNCS, vol. 12699, pp. 415–432. Springer (2021)
[36] Vukmirović, P., Bentkamp, A., Nummelin, V.: Efficient full higher-order unification. In: Ariola, Z.M. (ed.) FSCD. LIPIcs, vol. 167, pp. 5:1–5:17. Schloss Dagstuhl (2020)
[37] Vukmirović, P., Blanchette, J.C., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic. In: Vojnar, T., Zhang, L. (eds.) TACAS. LNCS, vol. 11427, pp. 192–210. Springer (2019)
[38] Vukmirović, P., Nummelin, V.: Boolean reasoning in a higher-order superposition prover. In: Fontaine, P., Korovin, K., Kotsireas, I.S., Rümmer, P., Tourret, S. (eds.) PAAR+SC^2. CEUR Workshop Proceedings, vol. 2752, pp. 148–166. CEUR-WS.org (2020)