Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Making Higher-Order Superposition Work

Petar Vukmirović · **Alexander Bentkamp** ·
Jasmin Blanchette · **Simon Cruanes** ·
Visa Nummelin · **Sophie Tourret**

Received: date / Accepted: date

Abstract Superposition is among the most successful calculi for first-order logic. Its extension to higher-order logic introduces new challenges such as infinitely branching inference rules, new possibilities such as reasoning about Booleans, and the need to curb the explosion of specific higher-order rules. We describe techniques that address these issues and extensively evaluate their implementation in the Zipperposition theorem prover. Largely thanks to their use, Zipperposition won the higher-order division of the CASC-J10 competition.

1 Introduction

Superposition-based first-order automatic theorem provers have emerged as useful reasoning tools. They dominate at the annual CASC [47] theorem prover competition, having always won the first-order theorem division. They are also used as backends to proof assistants [13, 26, 37], automatic higher-order theorem provers [44], and software verifiers [17].

The superposition calculus has only recently been extended to higher-order logic (more precisely, extensional simple type theory [20]), resulting in *Boolean-free λ-superposition* [6], which we developed together with Waldmann, as well as *combinatory superposition* [10] by Bhayat and Reger. Although these two calculi do not support an interpreted Boolean type, they can be extended by ad hoc rules [55] that support most of the Boolean reasoning necessary in practice.

Petar Vukmirović (✉) · Alexander Bentkamp · Jasmin Blanchette · Visa Nummelin
Vrije Universiteit Amsterdam, Amsterdam, the Netherlands
E-mail: {p.vukmirovic,a.bentkamp,j.c.blanchette,visa.nummelin}@vu.nl

Jasmin Blanchette · Sophie Tourret
Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
E-mail: {jasmin.blanchette,sophie.tourret}@inria.fr

Jasmin Blanchette · Sophie Tourret
Max-Planck-Institut für Informatik, Saarbrücken, Germany
E-mail: {jblanche,stourret}@mpi-inf.mpg.de

Simon Cruanes
Imandra, Austin, Texas, USA
E-mail: simon@imandra.ai

Both higher-order superposition calculi were designed to gracefully extend first-order reasoning. As most steps in higher-order proofs tend to be essentially first-order, extending the most successful first-order calculus to higher-order logic seemed worth trying. Our first attempt at testing this idea was in 2019: Zipperposition 1.5, based on Boolean-free λ-superposition, finished third in the higher-order theorem division of CASC-27 [49], 12 percentage points behind the winner, the tableau prover Satallax 3.4 [11].

Studying the competition results, we found that higher-order tableaux have some advantages over higher-order superposition. To bridge the gap, we developed techniques and heuristics that simulate tableaux in the context of saturation. We implemented them in Zipperposition 2, which took part in CASC-J10 [50] in 2020. This time, our prover won the division, proving 84% of the problems, a whole 20 percentage points ahead of the runner-up, Satallax 3.4.

In this article, we describe the main techniques that explain this reversal of fortunes. They cover most parts of a modern higher-order theorem prover, from preprocessing to additional calculus rules to heuristics to backend integration. We use a newer version of Zipperposition, based on a newer calculus: Instead of Boolean-free λ-superposition augmented with ad hoc Boolean rules, we work with *Boolean λ-superposition* [5], a principled extension of superposition to full higher-order logic, including an interpreted Boolean type.

Many higher-order problems extensively use symbol definitions to simplify their representation. We describe several ways to exploit the definitions, such as turning them into rewrite rules (Sect. 3). By working on formulas rather than clauses, tableau techniques take a more holistic view of a higher-order problem. Through its support for delayed clausification and, more generally, calculus-level formula manipulation, Boolean λ-superposition enables us to simulate most successful tableau techniques in a saturating prover (Sect. 4). This calculus also supports *Boolean selection functions*, a mechanism that allows us to choose on which Boolean subterms to perform inferences first. We implemented some Boolean selection functions and evaluated them (Sect. 5).

The main drawback of both λ-superposition variants compared with combinatory superposition is that they rely on rules that enumerate possibly infinite sets of unifiers. We describe a mechanism that interleaves infinitely branching inferences with the standard saturation process (Sect. 6). The prover retains the same behavior as before on first-order problems, smoothly scaling with increasing numbers of higher-order clauses. We also propose some heuristics to curb the explosion induced by highly prolific calculus rules (Sect. 7).

Using first-order backends to finish the proof is common practice in higher-order reasoning. Since λ-superposition coincides with standard superposition on first-order clauses, invoking backends may seem redundant; yet Zipperposition is nowhere as efficient as E [40] or Vampire [29], so invoking a more efficient backend does make sense. We describe how to achieve a balance between allowing native higher-order reasoning and delegating reasoning to a backend (Sect. 8). Finally, we compare Zipperposition 2 with other provers on all monomorphic higher-order TPTP benchmarks [48] to perform a more extensive evaluation than at CASC (Sect. 9). Our evaluation corroborates the competition results.

This article is an extended version of a paper accepted at CADE-28 [53]. Compared with the conference paper, it describes a new preprocessing technique, explores the effects of Boolean selection functions, evaluates more techniques, introduces new benchmark sets, and presents more examples.

2 Background and Setting

We focus on monomorphic higher-order logic, without the axiom of infinity or the axiom of at least two individuals. However, the techniques can easily be extended with polymorphism. Indeed, Zipperposition already supports some of them polymorphically.

Higher-Order Logic We define terms s,t,u,v inductively as free variables F,X, bound variables $x,y,z,\ldots$, constants $\mathsf{f},\mathsf{g},\mathsf{a},\mathsf{b},\ldots$, term applications $s\,t$, and λ-abstractions $\lambda x.\, s$. The syntactic distinction between free and bound variables yields *loose bound variables* (e.g., y in $\lambda x.\, y\,\mathsf{a}$) [33]. We let $s\,\bar{t}_n$ stand for $s\,t_1\,\ldots\,t_n$ and $\lambda\bar{x}_n.\, s$ for $\lambda x_1.\ldots.\lambda x_n.\, s$. The n-fold application of a unary term s to a term t is denoted by $s^n\,t$. Every β-normal term can be written as $\lambda\bar{x}_m.\, s\,\bar{t}_n$, where s is not an application; we call s the *head* of the term. If the type of a term t is of the form $\tau_1 \to \cdots \to \tau_n \to o$, where o is the distinguished Boolean type and $n \geq 0$, we call t a *predicate*. A term of type o is called a *formula*.

A literal l is an equation $s \approx t$ or a disequation $s \not\approx t$. A clause is a finite multiset of literals, interpreted and written disjunctively $l_1 \vee \cdots \vee l_n$. Logical symbols that may occur within terms are written in boldface: $\neg,\wedge,\vee,\rightarrow,\leftrightarrow,\ldots$. Quantified formulas are expressed using (a type-indexed family of) constants $\forall$ and $\exists$ as $\forall\,(\lambda x.\, t)$ and $\exists\,(\lambda x.\, t)$, usually abbreviated to $\forall x.\, t$ and $\exists x.\, t$. Following Boolean λ-superposition, predicate literals are encoded as equations with $\top$ or $\bot$: for example, $\mathsf{even}(x)$ becomes $\mathsf{even}(x) \approx \top$, and $\neg\,\mathsf{even}(x)$ becomes $\mathsf{even}(x) \approx \bot$.

Higher-Order Calculi The Boolean λ-superposition calculus [5] is a refutationally complete inference system and redundancy criterion for higher-order logic with rank-1 polymorphism, Hilbert choice, and functional and Boolean extensionality. The calculus relies on *complete sets of unifiers* (*CSUs*). The CSU for s and t with respect to a finite set of variables V, denoted by $\mathrm{CSU}_V(s,t)$, is a set of unifiers of s and t such that for any unifier ϱ of s and t, there exist substitutions $\sigma \in \mathrm{CSU}_V(s,t)$ and θ such that $\varrho(X) = \sigma(\theta(X))$ for all variables $X \in V$. The set V is used to distinguish important variables from auxiliary variables (which may arise in intermediary states of the unification procedure). We usually omit it.

Unlike Boolean-free λ-superposition, this calculus does not require axioms defining the logical symbols to cope with formulas. Instead, it includes Boolean inference rules that mimic superposition from such axioms into Boolean subterms, while avoiding the explosion incurred by adding these axioms to the proof state. It also includes rules that simulate Boolean inferences below applied variables. Both sets of rules are disabled or replaced with incomplete, ad hoc rules described by Vukmirović and Nummelin [55] in most configurations of the CASC portfolio. A new feature of the calculus that we explore in detail is the ability to select Boolean subterms to restrict Boolean and superposition inferences.

In contrast to both λ-superposition variants, combinatory superposition can avoid enumerating CSUs by using a form of first-order unification. Essentially, it enumerates higher-order terms using rules that instantiate applied variables with partially applied combinators from the complete combinator set $\{\mathsf{S},\mathsf{K},\mathsf{B},\mathsf{C},\mathsf{I}\}$. This calculus is the basis of Vampire 4.5 [10], which finished closely behind Satallax 3.4 at CASC-J10.

A different, very successful calculus is Satallax's SAT-guided tableaux [2]. Satallax was the leading higher-order prover of the 2010s. Its simple and elegant tableaux avoid deep superposition-style rewriting inferences. Nevertheless, our working hypothesis for the past six years has been that superposition would likely provide a stronger basis for higher-order reasoning. Other competing higher-order calculi include SMT (implemented in CVC4 [3,4]) and extensional paramodulation (implemented in Leo-III [44]).

Zipperposition Zipperposition [6, 12] is a higher-order theorem prover that implements both λ-superposition variants, combinatory superposition, and other superposition-like calculi. The prover was conceived as a testbed for rapidly experimenting with extensions of first-order superposition, but over time it has assimilated many of E's techniques and heuristics and become quite powerful.

Several of our techniques extend the *given clause procedure*, the standard saturation procedure pioneered by McCune and Wos [31, Sect. 2.3]. It partitions the proof state into a set P of *passive* clauses and a set A of *active* clauses. Initially, P contains all input clauses, and A is empty. At each iteration, a *given* clause is moved from P to A (i.e., it is *activated*), all inferences between it and clauses in A are performed, and the conclusions are added to P. Because Zipperposition fully simplifies clauses only when they are activated, it implements a DISCOUNT-style loop [14].

Experimental Setup To assess our techniques, we carried out experiments with Zipperposition 2. We used two sets of benchmarks: all 2851 monomorphic higher-order problems from the TPTP library [48] version 7.4.0 (labeled *TPTP*) and 1253 Sledgehammer-generated monomorphic higher-order problems (labeled *SH*). Although some techniques support polymorphism, we uniformly used monomorphic benchmarks.

We fixed a *base* configuration of Zipperposition parameters as a baseline for all comparisons. This is an incomplete, pragmatic configuration of Boolean λ-superposition using heuristics expected to perform well on a wide range of problems. In each experiment, we varied the parameters associated with a specific technique to evaluate it. The experiments were run on StarExec Miami [45] servers, equipped with Intel Xeon E5-2620 v4 CPUs clocked at 2.10 GHz. Unless otherwise stated, we used a CPU time limit of 15 s, roughly the time each configuration is given in the CASC portfolio mode. The raw evaluation results are available online.[1]

3 Preprocessing Higher-Order Problems

The TPTP library contains thousands of higher-order problems. Despite their diversity, they have a markedly different flavor from the TPTP first-order problems. Notably, they extensively use the `definition` role to identify universally quantified equations (and equivalences) that define symbols. Definitions $s \approx t$ (or $(s \leftrightarrow t) \approx \top$) can be replaced by rewrite rules $s \to t$, using the orientation given in the input problem. If there are multiple definitions for the same symbol, only the first one is replaced by a rewrite rule. Then, whenever a clause is picked in the given clause procedure, it will be rewritten using the collected rules. Alternatively, we can rewrite the input formulas as a preprocessing step. This ensures that the input clauses will be fully simplified when the proving process starts and no defined symbols will occur in clauses, which usually helps the heuristics.

Since the TPTP format enforces no constraints on definitions, rewriting might diverge. To ensure termination, we limit the number of applied rewrite steps. In practice, most TPTP problems are well behaved: Only one definition is given for each symbol, and the definitions are acyclic.

Turning the defining equations into rewrite rules, unfolding the definitions, and β-reducing the result can eliminate all of a problem's higher-order features, making it amenable to first-order methods. However, this can inflate the problem beyond recognition and compromise the refutational completeness of superposition.

[1] `http://doi.org/10.5281/zenodo.5007440`

Example 1 Removing higher-order features of a problem can have adverse effects. Consider the TPTP problem `NUM636^3`, which defines the predicate m as $\lambda x.\, \mathsf{s}\, x \not\approx x$ and states its conjecture as $\forall x.\, \mathsf{m}\, x$, where s is the standard Peano-style natural number successor constructor. When this definition is kept as is, the prover can superpose from either m or its definition into the (clausified) induction axiom, which is also given in the problem, and quickly prove the conjecture, without using any advanced inductive reasoning. In contrast, when the definition is unfolded and the problem is β-reduced, both m and the corresponding λ-abstraction disappear, forcing the prover to guess the correct instantiation for the induction axiom.

We describe two techniques to mitigate these issues. The first one is based on the observation that in practice, the explosion associated with definition unfolding mostly manifests itself on definitions of nonpredicate symbols. In some cases, it is preferable to rely on superposition's term order and powerful simplification engine to rewrite the proof state rather than to blindly rewrite definitions. On the other hand, superposition's reasoning with equivalences is often inadequate [5, 18]. Thus, it makes sense to treat only predicate definitions as rewrite rules.

The second technique aims at preserving completeness: We can try to force the term order that parameterizes superposition to orient as many definitions as possible and rely on demodulation to simplify the proof state. Usually, the Knuth–Bendix order (KBO) [27] is used. It compares terms by first comparing their weights, which is the sum of all the weights assigned to the symbols it contains. Given a symbol weight assignment $\mathscr{W}$, we can update it so that it orients acyclic definitions from left to right assuming that they are of the form $\mathsf{f}\, \overline{X}_m \approx \lambda \overline{Y}_n.\, t$, where the only free variables in t are $\overline{X}_m$, no free variable is repeated or appears applied in t, and f does not occur in t. Then we traverse the symbols f that are defined by such equations following the dependency relation, starting with a symbol f that does not depend on any other defined symbol. For each f, we set $\mathscr{W}(\mathsf{f})$ to $w+1$, where w is the maximum weight of the right-hand sides of f's definitions, computed using $\mathscr{W}$. By construction, for each equation the left-hand side is heavier. Thus, the equations are orientable from left to right.

Example 2 Many of the problems in the TPTP library's `LCL` category encode modal logic in higher-order logic. More complex modal operators (such as implication and equivalence) are defined in terms of basic connectives (such as negation and disjunction). Some of the definitions present in the problems are $\mathsf{mnot} := \lambda p\, x.\, \neg\, p\, x$, $\mathsf{mor} := \lambda p\, q\, x.\, p\, x \vee q\, x$, and $\mathsf{mimplies} = \lambda p\, q.\, \mathsf{mor}\, (\mathsf{mnot}\, p)\, q$. Assuming that the weight of λ, bound variables, and basic connectives is 2, we can orient equations using the above described approach as follows. Starting from symbols that do not depend on the other ones, we set $\mathscr{W}(\mathsf{mnot}) = 11$ and $\mathscr{W}(\mathsf{mor}) = 17$. Then, we use these values to set $\mathscr{W}(\mathsf{mimplies}) = 37$. Clearly, these weights enable us to orient all definitions from left to right.

Evaluation and Discussion We designed and evaluated the following strategies for handling `definition` axioms:

pre-RW	rewrite all definitions as a preprocessing step;
in-RW	rewrite all definitions during the saturation, as an inprocessing step;
o-RW	rewrite only predicate definitions, during preprocessing;
o-RW+KBO	like *o*-RW but with adjusted KBO weights for the remaining definitions;
no-RW	no special treatment of definitions;
no-RW+KBO	like *no*-RW but adjusting KBO weights for all definitions.

	pre-RW	in-RW	*o*-RW	*o*-RW+KBO	no-RW	no-RW+KBO
TPTP	**1635***	1619	1620	1621	1298	1296

Fig. 1: Impact of the definition rewriting method

The results are given in Fig. 1. In all the figures, each cell gives the number of proved problems, and cells marked with $\star$ correspond to the base configuration. The highest number in a category is typeset in bold. SH benchmarks are not included because they do not contain the `definition` role.

The four configurations in which definitions are treated as rewrite rules perform much better than the other two. In contrast, adjusting KBO weights gives no substantial improvement: Looking at raw data, we found only 2 problems proved by *o*-RW+KBO but not by *o*-RW, and in which the feature was used in the proof. For no-RW and no-RW+KBO, the 2-problem difference may be just noise. Even though it proves fewer problems, the configuration *o*-RW has some advantages over pre-RW: It proves 16 problems that pre-RW does not, 3 of which have a TPTP difficulty rating (the ratio of eligible provers that cannot prove the problem) of 1.

Rewriting after clausification avoids getting stuck rewriting parts of the proof state that might not contribute to the proof. In practice, we noticed that rewriting can be so expensive that the prover can spend all allotted CPU time in the preprocessing phase. The evaluation results confirm this observation: There are 64 problems proved by in-RW but not by pre-RW. Moreover, there are 41 problems that can be proved only by in-RW but not by any other above described configuration.

4 Reasoning about Formulas

Higher-order logic identifies formulas with terms of Boolean type. To prove a problem, we often need to instantiate a variable with the right predicate. Finding this predicate can be easier if the problem is not clausified. Consider the conjecture $\exists f.\, f\,\mathsf{p}\,\mathsf{q} \leftrightarrow \mathsf{p} \wedge \mathsf{q}$. Expressed in this form, the formula is easy to prove by taking $f := \lambda x\, y.\, x \wedge y$. By contrast, guessing the right instantiation for the negated, clausified form $F\,\mathsf{p}\,\mathsf{q} \approx \bot \vee \mathsf{p} \approx \bot \vee \mathsf{q} \approx \bot, F\,\mathsf{p}\,\mathsf{q} \approx \top \vee \mathsf{p} \approx \top$, $F\,\mathsf{p}\,\mathsf{q} \approx \top \vee \mathsf{q} \approx \top$ is more challenging. One of the strengths of higher-order tableau provers is that they do not clausify the input problem. This might partly explain Satallax's dominance in the THF division of CASC competitions until CASC-J10.

The Boolean λ-superposition calculus supports *delayed clausification rules* that insert problems into the proof state in their original, nonclausified form, and clausify them gradually. Delayed clausification allows the prover to analyze the syntactic structure of formulas during saturation, whereas the more traditional approach of *immediate clausification* applies a standard clausification algorithm [34] both as a preprocessing step and whenever predicate variables are instantiated.

An earlier evaluation of the Boolean λ-superposition calculus [5] showed that the *outer* variant of delayed clausification substantially increases this calculus's performance. The outer variant clausifies top-level logical symbols, proceeding from the outside inwards; for example, a clause $C \vee (\mathsf{p} \wedge \mathsf{q}) \approx \bot$ is transformed into $C \vee \mathsf{p} \approx \bot \vee \mathsf{q} \approx \bot$. The calculus also supports *inner* delayed clausification, which uses only the core calculus rules to clausify problems. Even though this is the laziest approach to clausification, the earlier evaluation showed that this approach is inefficient. Thus, we focus only on the outer rules.

Delayed clausification rules can be used as inference rules (which add conclusions to the passive set) or as simplification rules (which delete premises and add conclusions to the passive set). Inferences give more flexibility, since all intermediate clausification states will be stored in the proof state, at the cost of producing many clauses. Simplifications produce fewer clauses, but risk destroying informative syntactic structure. Since clausifying equivalences can destroy a lot of syntactic structure [18], we never apply simplifying rules on them.

Delayed clausification can interfere with clause splitting techniques. Zipperposition supports a lightweight variant of AVATAR [52], an architecture that partitions the search space by splitting clauses into variable-disjoint subclauses. This lightweight AVATAR is described by Ebner et al. [15, Sect. 7]. Combining it with delayed clausification makes it possible to split a clause $(\varphi_1 \vee \cdots \vee \varphi_n) \approx \top$, where the φ_i's are arbitrarily complex formulas that share no free variables with each other, into clauses $\varphi_i \approx \top$. To finish the proof, it suffices to derive the empty clause under each assumption $\varphi_i \approx \top$. Since the split is performed at the formula level, this technique resembles tableaux, but it exploits the strengths of superposition, such as its powerful redundancy criterion and simplification machinery, to close the branches.

Beyond splitting, interleaving clausification and saturation allows us to simulate another tableau-inspired technique. Whenever dynamic clausification substitutes a fresh variable X for a predicate variable x in a clause of the form $(\forall x.\, \varphi) \approx \top \vee C$, yielding $\varphi\{x \mapsto X\} \approx \top \vee C$, we can create additional clauses in which x is replaced with $t \in \mathit{Inst}$, where *Inst* is a set of heuristically chosen terms. This set contains λ-abstractions whose bodies are formulas and that occur in activated clauses, and *primitive instantiations* [55]—that is, imitations (in the sense of higher-order unification) of logical symbols that approximate the shape of a predicate that can instantiate a predicate variable.

Since a new term t can be added to *Inst* after a clause with a quantified variable of t's type has been activated, we remember the clauses $\varphi\{x \mapsto X\} \approx \top \vee C$ and instantiate them when *Inst* is extended. Conveniently, these instantiated clauses are not recognized as subsumed by Zipperposition, which uses an optimized, incomplete higher-order subsumption algorithm.

Given a disequation $\mathsf{f}\,\overline{s}_n \not\approx \mathsf{f}\,\overline{t}_n$, the *abstraction* of s_i is $\lambda x.\, u \not\approx v$, where u is obtained by replacing all occurrences of s_i in $\mathsf{f}\,\overline{s}_n$ with x and v is obtained by replacing all occurrences of s_i in $\mathsf{f}\,\overline{t}_n$ with x. For an equation $\mathsf{f}\,\overline{s}_n \approx \mathsf{f}\,\overline{t}_n$, the analogous abstraction is $\lambda x.\, \neg(u \not\approx v)$. Adding abstractions of the literals occurring in the conjecture to *Inst* can provide useful instantiations for formulas such as induction principles of datatypes. As the conjecture is negated in refutational theorem proving, the equation's polarity is inverted in the abstraction.

Example 3 The clausified conjecture of the problem `DAT056^2` [46] from the TPTP library is $\mathsf{ap}\,\mathsf{xs}\,(\mathsf{ap}\,\mathsf{ys}\,\mathsf{zs}) \not\approx \mathsf{ap}\,(\mathsf{ap}\,\mathsf{xs}\,\mathsf{ys})\,\mathsf{zs}$, where ap is the list append operator defined recursively on its first argument and xs, ys, and zs are of list type. Abstracting xs from the disequation yields $t = \lambda xs.\, \mathsf{ap}\,xs\,(\mathsf{ap}\,\mathsf{ys}\,\mathsf{zs}) \not\approx \mathsf{ap}\,(\mathsf{ap}\,xs\,\mathsf{ys})\,\mathsf{zs}$, which is added to *Inst*. Included in the problem is the induction axiom for the list datatype: $\forall p.\, p\,\mathsf{nil} \wedge (\forall x\,xs.\, p\,xs \to p\,(\mathsf{cons}\,x\,xs)) \to \forall xs.\, p\,xs$, where nil and cons have the usual meanings. Instantiating p with t and using the ap definition, we can prove $\forall xs.\, \mathsf{ap}\,xs\,(\mathsf{ap}\,\mathsf{ys}\,\mathsf{zs}) \not\approx \mathsf{ap}\,(\mathsf{ap}\,xs\,\mathsf{ys})\,\mathsf{zs}$, from which we easily derive a contradiction.

Evaluation and Discussion The base configuration (*base*) uses immediate clausification (IC) and disables lightweight AVATAR (−LA). To test the merits of delayed clausification, we vary *base*'s parameters along two axes: We choose immediate clausification (IC), delayed clausification as inference (DCI), or delayed clausification as simplification (DCS), and we either enable (+LA) or disable (−LA) lightweight AVATAR. Neither of the configurations uses instantiation with terms from *Inst*.

		+LA	−LA
TPTP	IC	1616	1635*
	DCI	1507	1532
	DCS	1668	**1703**
SH	IC	425	452*
	DCI	362	385
	DCS	441	**457**

Fig. 2: Impact of clausification and lightweight AVATAR

Figure 2 shows that using delayed clausification as simplification greatly increases the success rate, regardless of whether lightweight AVATAR is used. Using delayed clausification as inference has the opposite effect on both problem sets, presumably due to the large number of clauses it creates. By manually inspecting the proofs found by the DCS configuration, we noticed that a main reason for its success is that it does not simplify away equivalences. Overall, the lightweight AVATAR harms performance, but the sets of problems proved with and without it are vastly different. For example, the IC+LA configuration proves 38 problems not proved by IC−LA (i.e., *base*) on TPTP benchmarks and 14 such problems on SH benchmarks.

The Boolean instantiation technique presented above requires delayed clausification. We assessed it in the best configuration from Fig. 2, DCS−LA. With this change (+BI), Zipperposition proves 1700 TPTP problems and 456 SH problems. On TPTP, even though +BI solves 3 problems less than DCS−LA, it is very useful: 41 problems can be proved with +BI but not with DCS−LA. Conversely, 44 problems are solved with DCS−LA, but not with +BI, which suggests that Boolean instantiation can be explosive. One of the problems Boolean instantiation helps solve is `NUM636^2` (a re-encoding of `NUM636^3`). It conjectures that $\forall x.\, \mathsf{s}\, x \not\approx x$, where x ranges over Peano-style numbers specified by z and s. The given axioms are the induction principle $\forall p.\, p\, \mathsf{z} \wedge \forall x.\, (p\, x \rightarrow p\, (\mathsf{s}\, x)) \rightarrow \forall x.\, p\, x$, injectivity $\forall xy.\, \mathsf{s}\, x \approx \mathsf{s}\, y \rightarrow x \approx y$, and distinctness $\forall x.\, \mathsf{s}\, x \not\approx \mathsf{z}$. The conjecture is easily proved if Boolean instantiation is enabled: Even though the conjecture literal cannot be abstracted, instantiating p with the term $\lambda x.\, \mathsf{s}\, x \not\approx x$ used in the encoding of the (non-clausified) conjecture leads to a proof in just 22 given clause loop iterations. Zipperposition also finds a proof using the DCI−LA configuration, but this requires 294 iterations.

The +BI configuration proves 18 TPTP problems no other configuration from Fig. 2 can prove. Among these is `DAT056^2` (Example 3). In contrast, on SH benchmarks, only 6 problems are proved using +BI and not DCS−LA. For all these problems, Boolean instantiation does not appear in the proof, suggesting that this result is due to the randomness in the evaluation environment. The fact that BI has no effect on SH benchmarks is to be expected because Sledgehammer does not include lemmas whose name contains the substring `.induct` and that contain predicate variables. Therefore, BI applies to fewer clauses.

5 Exploring Boolean Selection Functions

Superposition calculi are parameterized by a literal selection function and a term order that help prune considerable swaths of the search space without jeopardizing completeness. The core inferences apply only to a clause's *eligible* literals, defined as either the clause's selected literals or, if none are selected, the clause's literals that are maximal with respect to the term

order. To further restrict which terms can be targeted by an inference, the Boolean λ-superposition calculus introduces *Boolean selection functions*.

A Boolean selection function chooses *green subterms* of Boolean type (different than $\top$ or $\bot$ and not occurring at either side of a positive literal) in a clause and gives rise to a notion of eligibility that considers the formula structure. Green subterms correspond to the first-order skeleton of a higher-order term; that is, they do not occur in positions under applied variables, quantifiers, or λ-abstractions.

Definition 4 (Green subterms and green positions) Green subterms and green positions are defined inductively as follows: t is a green subterm of t at green position ε; if t is a green subterm of u_i at green position p and f is a constant different from $\forall$ and $\exists$, then t is a green subterm of $\mathsf{f}\,\overline{u}_n$ at green position $i.p$, assuming $i \leq n$.

Example 5 The green subterms of the term $F\,\mathsf{a} \wedge \mathsf{p}\,(\forall\,(\lambda x.\,\mathsf{q}\,x))\,\mathsf{b}$ are the term itself, $F\,\mathsf{a}$, $\mathsf{p}\,(\forall\,(\lambda x.\,\mathsf{q}\,x))\,\mathsf{b}$, $\forall\,(\lambda x.\,\mathsf{q}\,x)$, and b.

Green positions are lifted to clauses as follows: If p is the green position of a subterm in s, and s occurs in a literal $l \in \{s \approx t,\, s \not\approx t\}$ of C, the green position of the same subterm in the clause is denoted by $l.s.p$. Boolean λ-superposition mandates additional restrictions on the Boolean selection function: $\top$, $\bot$ and variable-headed terms cannot be selected; for literals $s \approx t$, neither s not t cannot be selected; if a term s contains a variable X as a green subterm, and $X\,\overline{u}_n$, with $n \geq 1$, is a maximal term of the clause, s cannot be selected.

Definition 6 (Eligibility) Given a substitution σ and term order $\succ$, we say a literal l is (strictly) eligible with respect to σ in C if it is selected in C or there are no selected literals and no selected Boolean subterms in C and $l\sigma$ is (strictly) maximal in $C\sigma$ with respect to the term order. The eligible subterms of a clause C with respect to a substitution σ are inductively defined as follows: Any subterm selected by the Boolean selection function is eligible. For a strictly eligible literal $s \approx t$ with $t\sigma \not\succ s\sigma$, s is eligible. For an eligible literal $s \not\approx t$ with $t\sigma \not\succ s\sigma$, s is eligible. If a subterm t is eligible and the head of t is not $\approx$ or $\not\approx$, all direct green subterms of t are eligible. If a subterm t is eligible and t is of the form $u \approx v$ or $u \not\approx v$, then u is eligible if $v\sigma \not\succ u\sigma$ and v is eligible if $u\sigma \not\succ v\sigma$.

The above definitions of green subterms and eligibility were originally introduced with Boolean λ-superposition [5]. The Boolean selection function plays a similar role as the literal selection function in standard superposition. Literal selection functions eliminate some of the nondeterminism present in the superposition calculus by focusing on selected parts of the search space. Boolean selection functions achieve the same goal, but in a different context: They eliminate nondeterminism that is not present in standard superposition, namely, the choice of subformula on which the Boolean calculus rules are to be applied. As with literal selection functions, selecting few (and smaller) subterms can give rise to fewer possible inferences and reduce clause proliferation.

This notion of eligibility opens up possibilities for reasoning with formulas that are hard to simulate with the existing superposition machinery. For example, given a formula $\varphi \rightarrow \psi$, selecting the antecedent simulates forward reasoning whereas selecting the consequent simulates backward reasoning. The new eligibility also makes it possible to restrict the proof search to a small, promising part of a formula. Note that literal selection can override Boolean selection: Selecting a literal might make some of its green subterms eligible, regardless of Boolean selection.

In our previous work [35], we left this area of new possibilities largely unexplored. We designed simple functions that selected smallest, largest, innermost, or outermost terms, but

they did not impact performance much. Here we propose alternatives. Intuitively, a well-performing literal selection function might succeed at taming the combinatorial explosion if the selected literal can take part in few inferences [21]. However, Boolean selection functions introduce another factor to consider: the context in which the selected subterm occurs. This suggests the following definition:

Definition 7 (Contextualized Boolean selection function) Let $ctx(C)$ be a function that maps a clause C to a set of green positions p such that $C|_p$ is a selectable Boolean subterm, and let $\rhd$ be a partial order on pairs of terms and green positions. The *context Boolean selection function* $Sel^{\rhd}_{ctx}(C)$ selects all terms t such that $t = C|_p$, $p \in ctx(C)$, and (t,p) is maximal with respect to $\rhd$.

In the above definition, the function *ctx* lets us choose the context in which the Boolean subterm appears. Then, among the terms in the chosen context, we choose the ones that are maximal with respect to $\rhd$.

Ganzinger and Stuber considered Boolean subterm selection for their extension of first-order superposition with interpreted Boolean type [18]. Unlike our calculus, their calculus requires selection of subterms occurring in negative green positions, defined below.

Definition 8 (Polarity of green positions) Negative and positive green positions in a clause $C = l_1 \vee \cdots \vee l_n$ are defined inductively as follows: For each $1 \leq i \leq n$, the green position $l_i.s$ is positive if $l_i = s \approx \top$ and negative if $l_i = s \approx \bot$. If p is positive (negative) and $C|_p = s\,\bar{t}_n$ where s is either $\wedge$ or $\vee$, then each $p.i, 1 \leq i \leq n$, is positive (negative). If p is positive and $C|_p = \neg\, s$, then $p.1$ is negative; if p is negative and $C|_p = \neg\, s$, then $p.1$ is positive. If p is positive and $C|_p = s \to t$, then $p.1$ is negative and $p.2$ is positive; if p is negative and $C|_p = s \to t$, then $p.1$ is positive and $p.2$ is negative.

Note that the polarity of p is undefined whenever $C|_p$ is not a green Boolean subterm or it occurs under a (dis)equivalence or an uninterpreted symbol. To assess how the function *ctx* affects performance, we use the following selection functions that consider green positions of selectable Boolean terms:

`Any`	select all green positions;
`Pos`	select all positive green positions;
`Neg`	select all negative green positions;
`Forward`	select all green positions $p = q.1$ such that $C|_q = s \to t$;
`Backward`	select all green positions $p = q.2$ such that $C|_q = s \to t$;
`Deep`	select all green positions of maximal length;
`Shallow`	select all green positions of minimal length.

We also introduce three partial orders for selecting subterms from a given context. For all three orders, if exactly one of the subterms has a logical head, then the subterm with the nonlogical head is larger, because logical symbols are more explosive. Otherwise, the orders use the following criteria:

$\rhd_{\text{ground}}$ If exactly one of the subterms is ground, make the ground subterm larger; otherwise, if exactly one of the subterms is of the form $s \approx t$, make this subterm larger.

$\rhd_{\text{depth}}$ If one of the subterms has larger subterm depth (longest valid green subterm position), make this subterm larger; otherwise, if one of the subterms has less distinct variables, make this subterm larger.

		Any	Pos	Neg	Forward	Backward	Deep	Shallow
TPTP	$\rhd_{\text{ground}}$	1538	1550	1547	1534	**1554**	1539	1538
	$\rhd_{\text{depth}}$	1542	1550	1528	1542	1550	1547	1535
	$\rhd_{\text{def}}$	1543	1551	1540	1540	1551	1545	1537
SH	$\rhd_{\text{ground}}$	386	379	386	386	379	**387**	**387**
	$\rhd_{\text{depth}}$	377	376	384	378	376	379	376
	$\rhd_{\text{def}}$	379	374	**387**	379	380	377	381

Fig. 3: Impact of the Boolean selection function

$\rhd_{\text{def}}$ If exactly one of the subterms is of the form $\mathsf{p}\,\overline{X}_n$ where $\overline{X}_n$ is a tuple of free variables, make the other subterm larger; otherwise, if exactly one of the subterms is of the form $X\,\overline{s}_n$, make the other subterm larger.

In case of a tie, the subterm with the smaller syntactic weight is made larger, and if both subterms have the same weight, the term that occurs in a position further to the left (i.e., that has a lexicographically smaller position) is made larger.

These orders follow the design principle enunciated by Hoder et al. [21] that ground or deep terms and terms with repeated variables are "less unifiable" with the similar observation for higher-order logic that reasoning about interpreted symbols or applied variables is usually explosive.

Example 9 Selecting the right Boolean subterm can help avoid elaborating higher-order inferences. Consider the unsatisfiable clause set consisting of $\mathsf{p}\,(\lambda y.\, X\,(\lambda x.\, x)\,\mathsf{a}) \to \neg\,(\mathsf{p}\,(\lambda y.\, X\,y\,\mathsf{a}))$, $\mathsf{p}\,(\lambda y.\, \mathsf{a})$, and $\mathsf{p}\,(\lambda y.\, y^{100}\,\mathsf{b})$. Note that $\mathsf{p}\,(\lambda y.\, X\,(\lambda x.\, x)\,\mathsf{a})$ and $\mathsf{p}\,(\lambda y.\, \mathsf{a})$ have infinitely many unifiers of the form $\{X \mapsto \lambda f x.\, f^i\,(x)\}, i \geq 0$, whereas $\mathsf{p}\,(\lambda y.\, X\,y\,\mathsf{a})$ and $\mathsf{p}\,(\lambda y.\, y^{100}\,\mathsf{b})$ have only one unifier. If `Forward` context selection is enabled, $\mathsf{p}\,(\lambda y.\, X\,(\lambda x.\, x)\,\mathsf{a})$ is made the target of superposition inference, forcing computation of at least 100 unifiers (under the assumption that unifiers are returned in order of increasing i) before we get to refute $\neg\,(\mathsf{p}\,(\lambda y.\, y^{100}\,\mathsf{b}))$. In contrast, `Backward` context selection allows us to superpose from $\mathsf{p}\,(\lambda y.\, y^{100}\,\mathsf{b})$ into $\mathsf{p}\,(\lambda y.\, X\,y\,\mathsf{a})$, avoiding this explosion.

Evaluation and Discussion When the input problem is clausified using immediate clausification, almost all Boolean structure is lost. In this case, we expect Boolean selection to have a modest effect. To better assess this feature, in this evaluation we use DCI−LA from Sect. 4 as the baseline configuration. To avoid interference of literal and Boolean selection, we additionally forbid the literal selection function from selecting a literal if it contains a selectable Boolean subterm.

The results of evaluating 21 concrete selection functions obtained by instantiating the contextualized Boolean selection function are shown in Fig. 3. Rows denote the used partial order $\rhd$, while columns denote the function *ctx*.

On TPTP benchmarks, Boolean selection helps tame the explosion caused by dynamic clausification used as inference: All but one selection functions outperform the DCI−LA baseline of 1532 proved problems. Coming back to the problem `NUM636^2` from Sect. 4, using Boolean selection can reduce the number of given clause loop iterations from 294 to 71.

The results suggest that selection of term context has more impact than the partial term order. Also, the best results are obtained when a context more specific than `Any` is chosen. Remarkably, functions using `Pos` context perform better than the ones using `Neg` context on TPTP but the opposite is observed on SH.

Using different Boolean selection functions yields vastly different sets of proved problems on TPTP benchmarks: In total, there are 103 problems proved by some configuration from Fig. 3 but not by DCI−LA. However, there are only 16 such SH problems. It would seem that the advanced formula reasoning facilitated by the Boolean selection formulas is usually not required by Sledgehammer problems.

6 Enumerating Infinitely Branching Inferences

As an optimization and to simplify the code, Leo-III [42] and Vampire 4.4 [9] (which uses *restricted combinatory unification*, a predecessor of combinatory superposition) compute only a finite subset of the possible conclusions for inferences that require enumerating a CSU. Not only is this a source of incompleteness, but choosing the cardinality of the computed subset is a difficult heuristic choice. Small sets can result in missing the unifier necessary for the proof, whereas large sets make the prover spend too long in the unification procedure, generate useless clauses, and possibly get sidetracked into the wrong parts of the search space.

We propose a modification to the given clause procedure to seamlessly interleave unifier computation and proof state exploration. Given a complete unification procedure, which may yield infinite streams of unifiers, our modification fairly enumerates all conclusions of inferences relying on elements of a CSU. Under some reasonable assumptions, it behaves exactly like the standard given clause procedure on purely first-order problems. We also describe heuristics that help achieve a similar performance as when using incomplete, terminating unification procedures without sacrificing completeness.

Given that we cannot decide whether there exists a next CSU element in a stream of unifiers, the request for the next conclusion might not terminate, effectively bringing the theorem prover to a halt. Our modified given clause procedure expects the unification procedure to return a lazily computed stream [36, Sect. 4.2], where each element is either $\emptyset$ or a singleton set containing a unifier. To avoid getting stuck waiting for a unifier that may not exist, the unification procedure should return $\emptyset$ after it performs some number of operations without finding a unifier.

The complete unification procedure by Vukmirović et al. [56] returns such a stream. Other procedures such as Huet's [23] and Jensen and Pietrzykowski's [24] can easily be adapted to meet this requirement. Based on the stream of unifiers interspersed with $\emptyset$, we can construct a stream of inferences similarly interspersed with $\emptyset$. Any finite prefixes of this stream can be computed in finite time.

To support such streams in the given clause procedure, we extend it to represent the proof state not only by the active (A) and passive (P) clause sets, but also by a priority queue Q containing the inference streams, similar to the "to do" set T present in the abstract Zipperposition loop of Waldmann et al. [57, Sect. 4]. Each stream is associated with a weight, and Q is sorted in order of increasing weight, a low weight corresponding to a high priority. When they introduced λ-superposition, Bentkamp et al. [6] described an older version of this extension. Here we present a newer version in more detail, including heuristics to postpone unpromising streams. The pseudocode of the modified procedure is as follows:

function EXTRACTCLAUSE(Q, *stream*)
  maybe_clause ← pop and compute the first element of *stream*
  if *stream* is not empty **then**
    add *stream* to Q with an increased weight
  return *maybe_clause*

function HEURISTICPROBE(Q)
  $i \leftarrow 0$
  collected_clauses $\leftarrow \emptyset$
  while $i < K_{\text{best}}$ and $Q \neq \emptyset$ **do**
    $j \leftarrow 0$
    maybe_clause $\leftarrow \emptyset$
    while $j < K_{\text{retry}}$ and $Q \neq \emptyset$ and *maybe_clause* $= \emptyset$ **do**
      stream $\leftarrow$ pop the lowest-weight stream in Q
      maybe_clause $\leftarrow$ EXTRACTCLAUSE(Q, *stream*)
      $j \leftarrow j+1$
    collected_clauses $\leftarrow$ *collected_clauses* $\cup$ *maybe_clause*
    $i \leftarrow i+1$
  return *collected_clauses*

function FAIRPROBE(Q, *num_oldest*)
  collected_clauses $\leftarrow \emptyset$
  oldest_streams $\leftarrow$ pop *num_oldest* oldest streams from Q
  for *stream* in *oldest_streams* **do**
    collected_clauses $\leftarrow$ *collected_clauses* $\cup$ EXTRACTCLAUSE(Q, *stream*)
  return *collected_clauses*

function FORCEPROBE(Q)
  collected_clauses $\leftarrow \emptyset$
  while $Q \neq \emptyset$ and *collected_clauses* $= \emptyset$ **do**
    collected_clauses $\leftarrow$ FAIRPROBE(Q, $|Q|$)
  if $Q =$ *collected_clauses* $= \emptyset$ **then**
    status $\leftarrow$ Satisfiable
  else
    status $\leftarrow$ Unknown
  return (*status*, *collected_clauses*)

function GIVENCLAUSE(P, A, Q)
  $i \leftarrow 0$
  status $\leftarrow$ Unknown
  while *status* $=$ Unknown **do**
    if $P = \emptyset$ **then**
      (*status*, *forced_clauses*) $\leftarrow$ FORCEPROBE(Q)
      $P \leftarrow P \cup$ *forced_clauses*
    else
      given $\leftarrow$ pop a chosen clause from P and simplify it
      if *given* is the empty clause **then**
        status $\leftarrow$ Unsatisfiable
      else
        $A \leftarrow A \cup \{$*given*$\}$
        for *stream* in streams of inferences between *given* and *other* $\in A$ **do**
          if *stream* is not empty **then**
            $P \leftarrow P \cup$ EXTRACTCLAUSE(Q, *stream*)

$i \leftarrow i+1$
if $i \bmod K_{\text{fair}} = 0$ **then**
  $P \leftarrow P \cup \text{FAIRPROBE}(Q, i/K_{\text{fair}})$
else
  $P \leftarrow P \cup \text{HEURISTICPROBE}(Q)$
return *status*

Initially, all input clauses are put into P, and A and Q are empty. Unlike in the standard given clause procedure, inference results are represented as clause streams. The first element is inserted into P, and the rest of the stream is stored in Q with some positive integer weight computed from the inference rule.

To eventually consider inference conclusions from streams in Q as given clauses, we extract elements from, or *probe*, streams and move any obtained clauses to P. Analogously to the traditional pick–given ratio [31, 39], we use a parameter K_{fair} (by default, $K_{\text{fair}} = 70$) to ensure fairness: Every K_{fair}th iteration, FAIRPROBE probes an increasing number of oldest streams, which achieves dovetailing. In all other iterations, HEURISTICPROBE attempts to extract up to K_{best} clauses from the most promising streams (by default, $K_{\text{best}} = 7$). In each attempt, the most promising stream in Q is chosen. If its first element is $\emptyset$, the rest of the stream is inserted into Q and a new stream is chosen. This is repeated until either K_{retry} occurrences of $\emptyset$ have been met (by default, $K_{\text{retry}} = 20$) or the stream yields a singleton. Setting $K_{\text{retry}} > 0$ increases the chance that HEURISTICPROBE will return K_{best} clauses, as desired. Finally, if P becomes empty, FORCEPROBE searches relentlessly for a clause in Q, as a fallback.

The function EXTRACTCLAUSE extracts an element from a nonempty stream not in Q and inserts the remaining stream into Q with an increased weight, calculated as follows. Let n be the number of times the stream was chosen for probing. If probing results in $\emptyset$, the stream's weight is increased by $\max\{2, n-16\}$. If probing results in a clause C whose penalty is p, the stream's weight is increased by $p \cdot \max\{1, n-64\}$. The penalty of a clause is a number assigned by Zipperposition based on features such as the depth of its derivation and the rules used in it. The constants 16 and 64 increase the chance that newer clause-producing streams are picked, which is desirable because their first clauses are expected to be useful.

All three probing functions are invoked by GIVENCLAUSE, which contains the saturation loop. It differs from the standard given clause procedure in three ways: First, the proof state includes Q in addition to P and A. Second, new inferences involving the given clause are added to Q instead of being performed immediately. Third, inferences in Q are periodically performed lazily to fill P.

Example 10 Consider the unsatisfiable two-clause problem $\{X\,\mathsf{a} \not\approx \mathsf{f}\,(X\,\mathsf{a}) \vee \mathsf{p}\,(X\,\mathsf{a}),\ \neg\mathsf{p}\,(\mathsf{f}^{100}\,\mathsf{a})\}$ and a selection function which selects negative literals. Let $P \mid A \mid Q$ denote the state of the given clause loop (i.e., the contents of the passive and active set and of the stream queue), and let $[a_1, a_2, \ldots]$ denote an infinite stream of elements.

The given clause loop begins in the state $X\,\mathsf{a} \not\approx \mathsf{f}\,(X\,\mathsf{a}) \vee \mathsf{p}\,(X\,\mathsf{a}), \neg\mathsf{p}\,(\mathsf{f}^{100}\,\mathsf{a}) \mid \emptyset \mid \emptyset$. If the clause $\neg\mathsf{p}\,(\mathsf{f}^{100}\,\mathsf{a})$ is chosen for processing, since Q is empty and no inferences with the chosen clause are possible, the state becomes $X\,\mathsf{a} \not\approx \mathsf{f}\,(X\,\mathsf{a}) \vee \mathsf{p}\,(X\,\mathsf{a}) \mid \neg\mathsf{p}\,(\mathsf{f}^{100}\,\mathsf{a}) \mid \emptyset$. When the clause $X\,\mathsf{a} \not\approx \mathsf{f}\,(X\,\mathsf{a}) \vee \mathsf{p}\,(X\,\mathsf{a})$ is chosen, a new stream which enumerates results of equality resolution (on its first literal) is created. There are infinitely many conclusions of this inference, since there are infinitely many unifiers for the first literal of the form $\{X \mapsto \lambda x.\, \mathsf{f}^i\, x\}$, for $i \geq 0$. Thus, the stream is $[\{\mathsf{p}\,\mathsf{a}\}, \{\mathsf{p}\,(\mathsf{f}\,\mathsf{a})\}, \ldots]$, possibly with $\emptyset$s interspersed.

With the standard given clause procedure, there would have been no way to represent this infinitary result.

When the stream is created, its first element is popped and put into P. Then, based on the parameters that control inference stream probing, some number of clauses from the stream are computed and moved to P. After two iterations, the state might be $\mathsf{p}\,\mathsf{a}, \mathsf{p}\,(\mathsf{f}\,\mathsf{a}), \mathsf{p}\,(\mathsf{f}\,(\mathsf{f}\,\mathsf{a})) \mid X\,(\mathsf{f}\,\mathsf{a}) \not\approx \mathsf{f}\,(X\,\mathsf{a}) \vee \mathsf{p}\,(X\,\mathsf{a}), \neg\mathsf{p}\,(\mathsf{f}^{100}\,\mathsf{a}) \mid [\{\mathsf{p}\,(\mathsf{f}^{3}\,\mathsf{a})\}, \ldots]$.

In the next iterations, some clause of the form $\mathsf{p}\,(\mathsf{f}^{i}\,\mathsf{a})$, where $i < 100$, is chosen, but no inferences with it can be performed. Then the stream created in the second iteration is probed, and its results fill the set P. Ultimately, the clause $\mathsf{p}\,(\mathsf{f}^{100}\,\mathsf{a})$ is chosen, at which point $\bot$ is quickly derived.

GIVENCLAUSE eagerly stores the first element of a new inference stream in P to imitate the standard given clause procedure. If the underlying unification procedure behaves like the standard first-order unification algorithm on higher-order logic's first-order fragment, our given clause procedure coincides with the standard one. The unification procedure by Vukmirović et al. terminates on the first-order and other fragments [33]. To avoid computing complicated unifiers eagerly, it immediately returns $\emptyset$ for a problem that does not belong to one of the fragments that admit efficient unifier computation.

The design of our given clause procedure was guided by folklore knowledge about higher-order theorem proving. First, in our experience most steps in long higher-order proofs involve first-order literals. The unification procedure and inference scheduling ensure that first-order inference conclusions are put in the proof state as early as possible. Second, some inference rules are expected to be largely useless. We initialize the stream penalty differently for each rule, allowing old streams of more useful inferences to be queried before newly added, but potentially less useful streams. Finally, if we use a unification procedure that has aggressive redundancy elimination, we will often find the necessary unifier within the first few unifiers returned. Similarly, if a stream keeps returning $\emptyset$, it is likely that it is blocked in a nonterminating computation and should be ignored. Our heuristics to increase the stream penalties take both observations into account.

Evaluation and Discussion When the unification procedure of Vukmirović et al. was implemented in Zipperposition, it was observed that this is the only competing higher-order prover that proves all Church numeral problems from the TPTP, never spending more than 5 s on a problem [56]. On these hard unification problems, the stream system allows the prover to explore the proof state lazily.

Consider the TPTP problem `NUM800^1`, which requires finding a function F such that $F\,\mathsf{c}_1\,\mathsf{c}_2 \approx \mathsf{c}_2 \wedge F\,\mathsf{c}_2\,\mathsf{c}_3 \approx \mathsf{c}_6$, where c_n abbreviates the Church numeral for n, $\lambda s\,z.\, s^n\,z$. To prove the problem, it suffices to take F to be the multiplication operator $\lambda x\,y\,s\,z.\, x\,(y\,s)\,z$. However, this unifier is only one out of many available for each occurrence of F.

In an independent evaluation setup on a set of 2606 TPTP version 7.2.0 problems almost identical to the one we use, Vukmirović et al. [56, Sect. 7] compared a complete, nonterminating variant of the unification procedure and a pragmatic, terminating variant. The pragmatic variant was used directly—all the inference conclusions were put immediately in P, bypassing Q. The complete variant, which relies on possibly infinite streams and is much more prolific, proved only 15 problems less than the most competitive pragmatic variant. Furthermore, it proved 19 problems not proved by the pragmatic variant. This shows that our given clause procedure, with its heuristics, allows the prover to defer exploring less promising branches of the unification and uses the full power of a complete higher-order unifier search to solve unification problems that cannot be proved by a restricted procedure.

		K_{fair} = 2			K_{fair} = 16			K_{fair} = 128		
		K_{retry} = 2	16	128	2	16	128	2	16	128
K_{best}	2	1643	1645	1645	1661	1661	1658	1669	1664	1664
	16	1647	1646	1609	**1670**	1654	1602	1665	1659	1597
	128	1646	1644	1583	1661	1656	1577	1665	1658	1576

(a) TPTP benchmarks

		K_{fair} = 2			K_{fair} = 16			K_{fair} = 128		
		K_{retry} = 2	16	128	2	16	128	2	16	128
K_{best}	2	460	455	454	465	463	458	466	461	461
	16	458	453	445	464	459	441	**468**	459	442
	128	456	452	430	465	458	428	**468**	459	425

(b) SH benchmarks

Fig. 4: Impact of the stream enumeration parameter

The parameters K_{fair}, K_{retry}, and K_{best} can greatly influence the behavior of the given clause procedure, even when the same unification procedure is used. Figure 4 presents the effects of these parameters on TPTP and SH. Selecting a low number of best clauses seems to perform well on both benchmark sets. However, on SH benchmarks, which require overwhelmingly first-order unifiers, visiting older streams should be delayed a lot.

As with Boolean selection functions, changing these three parameters causes a substantial difference in the set of proved problems. For example, the configuration that performs the worst on TPTP bencharks proves 12 problems that the configuration performing the best on TPTP cannot prove; moreover, there are 29 TPTP problems that are proved by some set of parameters other than $K_{\text{fair}} = K_{\text{best}} = 16, K_{\text{retry}} = 2$. On SH, these effects are much weaker; most reasonable combinations of parameters perform similarly.

Among the competing higher-order provers, only Satallax uses infinitely branching calculus rules. It maintains a queue of "commands" that contain instructions on how to create a successor state in the tableau. One command describes infinite enumeration of all closed terms of a given function type. Each execution of this command makes progress in the enumeration. Unlike evaluation of streams representing elements of CSU, each command execution is guaranteed to make progress in enumerating the next closed functional term, so there is no need to ever return $\emptyset$.

7 Controlling Prolific Rules

To support higher-order features such as function extensionality and quantification over functions, many refutationally complete calculi employ highly prolific rules. For example, λ-superposition includes a FLUIDSUP rule [6] that very often applies to two clauses if one of them contains a term of the form $F\,\overline{s}_n$, where $n > 0$. We describe three mechanisms to keep rules like these under control.

First, *we limit applicability of the prolific rules*. In practice, it often suffices to apply prolific higher-order rules only to initial or shallow clauses—clauses with a shallow derivation depth. Thus, we added an option to forbid the application of a rule if the derivation depth of any premise exceeds a limit.

Second, *we penalize the streams of expensive inferences*. The weight of each stream is given an initial value based on characteristics of the inference premises such as their derivation depth. For prolific rules such as FLUIDSUP, we increment this value by a parameter K_{incr}. Weights for less prolific variants of this rule, such as DUPSUP [6], are increased by a fraction of K_{incr} (e.g., $\lfloor K_{\text{incr}}/3 \rfloor$).

Third, *we defer the selection of prolific clauses*. To select the given clause, most saturating provers evaluate clauses according to some criteria and choose the clause with the lowest evaluation. To make this choice efficient, passive clauses are organized into a priority queue ordered by their evaluations. Like E, Zipperposition maintains multiple queues, ordered by different evaluations, that are visited in a round-robin fashion. It also uses E's two-layer evaluation functions, a variant of which has recently been implemented in Vampire [19]. The two layers are *clause priority* and *clause weight*. Clauses with higher priority are preferred, and the weight is used for tie-breaking. Intuitively, the first layer crudely separates clauses into priority classes, whereas the second one uses heuristic weights to prefer clauses within a priority class. To control the selection of prolific clauses, we introduce new clause priority functions that take into account features specific to higher-order clauses.

The first new priority function, `PreferHOSteps` (`PHOS`), assigns a higher priority if rules specific to higher-order superposition calculi were used in the clause derivation. Since most of the other clause priority functions tend to defer higher-order clauses, having a clause queue that prefers them might be useful to find some proof more efficiently. A simpler function, which prefers clauses containing λ-abstractions, is `PreferLambda` (`PL`).

`PreferHOSteps` separates clauses created using first- and higher-order inference rules crudely. However, within higher-order inference rules there are the ones which make clauses simpler and are thus more preferable. An example of such a rule is

$$\frac{C \vee s \approx t}{C \vee s\,\overline{X}_n \approx t\,\overline{X}_n}\ \text{ARGCONG}$$

where s is of the type $\alpha_1 \to \cdots \to \alpha_k \to \beta$, β is a base type, $n \leq k$, free variables $\overline{X}_n$ are fresh, and literal $s \approx t$ is strictly eligible. When $n = k$, in most cases, the resulting clause has a first-order literal $s\,\overline{X}_n \approx t\,\overline{X}_n$ in place of the literal $s \approx t$ of functional type, which usually makes the clause more useful. To prefer clauses that are only mildly higher-order, we designed the function `PreferEasyHO` (`PEHO`). It prefers clauses that are the result of ARGCONG, have equations between terms of functional type or between higher-order patterns, or have literals containing logical symbols, in that order of priority.

A higher-order inference that applies a complicated substitution to a clause is usually followed by a $\beta\eta$-normalization step. If $\beta\eta$-normalization greatly reduces the size of a clause, it is likely that this substitution simplifies the clause (e.g., by removing a variable's arguments). The new priority function `ByNormalizationFactor` (`BNF`) is designed to exploit this observation. It prefers clauses that were produced by $\beta\eta$-normalization, and among those it prefers the ones with larger size reductions.

Another new priority function is `PreferShallowAppVars` (`PSAV`). This prefers clauses with lower depths of the deepest occurrence of an applied variable—that is, $C[X\,\mathsf{a}]$ is preferred over $C[\mathsf{f}\,(X\,\mathsf{a})]$. The intuition is that applying a substitution to an applied variable often reduces the variable to a term with a constant head, yielding a less explosive clause,

	CP	BAVN	PL	PSAV	PHOS	PEHO	BNF	PDAV
TPTP	1635*	**1640**	1604	1635	1609	1617	1575	1533
SH	**452***	451	417	450	439	407	411	302

Fig. 5: Impact of the priority function

	∞	16	8	4	2	1
TPTP	**1635***	1625	1632	1629	1628	1618
SH	**452***	438	435	439	435	440

Fig. 6: Impact of the FLUIDSUP weight increment K_{incr}

and the gain is greater for variables closer to the top level. Among the functions that rely on properties of applied variables, we implemented `PreferDeepAppVars` (`PDAV`), which returns the priority opposite of `PSAV`, and `ByAppVarNum` (`BAVN`), which prefers clauses with fewer occurrences of applied variables.

Evaluation and Discussion In the base configuration (*base*), Zipperposition visits several clause queues. The configuration uses queues that prefer the clauses that stem from the conjecture, the ones that have at least one positive literal, the ones that have been moved from active to passive set, and so on. One of the queues uses the constant priority function `ConstPrio` (`CP`), meaning that it assigns the same priority to every clause. As this queue is the most often visited one in *base*, changing its priority function should affect the result noticeably. To evaluate the new priority functions, we replaced `CP` with one of the new functions in this queue, leaving the clause weight intact. The results are shown in Fig. 5.

Even though constant priority function achieves remarkable performance, the new priority functions are useful additions to the prover's repertoire: 37 additional TPTP problems and 17 additional SH problems can be proved when some nonconstant priority is used. The generally average-performing `PEHO` function can prove 9 problems not proved with any other priority function on TPTP (and 1 on SH). Globally, 24 TPTP problems and 6 SH problems can be proved exclusively using one particular priority function.

Although it is necessary for refutational completeness, the FLUIDSUP rule is disabled in *base* because it is so explosive and so seldom useful. To test whether increasing inference stream weights makes a difference on the success rate, we tried enabling FLUIDSUP with different weight increments K_{incr} for FLUIDSUP inference queues. The results are shown in Fig. 6. As expected, using a low increment with FLUIDSUP is detrimental on TPTP. On this benchmark set, 16 additional problems can be proved when FLUIDSUP is enabled. The penalty mostly affects only proving time: All but 2 of these problems were proved by using at least three different values of K_{incr}. On SH problems, the best result is obtained when the rule is disabled as well. Unexpectedly, the next best result is obtained when $K_{\text{incr}} = 1$.

8 Controlling the Use of Backends

Cooperation with efficient off-the-shelf first-order theorem provers is an essential feature of higher-order theorem provers such as Leo-III [42, Sect. 4.4] and Satallax [11]. Those

	−Ehoh	0.1	0.25	0.5	0.75
TPTP	1635⋆	**1981**	1980	1979	1972
SH	452⋆	606	**608**	600	592

Fig. 7: Impact of the backend invocation point K_{time}

	−Ehoh	lifting	SKBCI	omitted
TPTP	1635⋆	**1980**	1877	1866
SH	452⋆	**608**	577	566

Fig. 8: Impact of the method used to translate λ-abstractions

provers invoke first-order backends repeatedly during a proof attempt and spend a substantial amount of time in backend collaboration. Since λ-superposition generalizes a highly efficient first-order calculus, we expect that future efficient λ-superposition implementations will not benefit much from backends. Nevertheless, experimental provers such as Zipperposition can still gain a lot. We present some techniques for controlling the use of backends.

In his thesis [42, Sect. 6.1], Steen extensively studies the effects of using different first-order backends on the performance of Leo-III. His results suggest that adding only one backend already substantially improves the performance. To reduce the effort required for integrating multiple backends, we chose Ehoh [54] as our single backend. Ehoh is an extension of the highly optimized superposition prover E 2.5 with support for higher-order features such as partial application, applied variables, and interpreted Booleans. On the one hand, Ehoh provides the efficiency of E while easing the translation from full higher-order logic—the only missing syntactic feature is λ-abstraction. On the other hand, Ehoh's higher-order reasoning capabilities are limited. Its unification algorithm is essentially first-order and it cannot synthesize λ-abstractions.

In a departure from Leo-III and other cooperative provers, instead of regularly invoking the backend, we invoke it at most once during a run of Zipperposition. This is because most competitive higher-order provers, including Zipperposition, use a portfolio mode in which many configurations are run for a short time, and we want to leave enough time for native higher-order reasoning. Moreover, multiple backend invocations tend to be wasteful, because currently each invocation starts with no knowledge of the previous ones.

Only a carefully chosen subset of the available clauses are translated and sent to Ehoh. Let I be the set of clauses representing the input problem. Given a proof state, let M denote the union of the current active and passive sets, and let M_{ho} denote the subset of M that contains only clauses that were derived using at least one λ-superposition rule not present in regular superposition. We order the clauses in M_{ho} by increasing derivation depth, using syntactic weight to break ties. Then we choose all clauses in I and the first K_{size} clauses from M_{ho} for use with the backend reasoner. We leave out clauses in $M \setminus (I \cup M_{\text{ho}})$ because Ehoh can rederive them. We also expect large clauses with deep derivations to be less useful.

The remaining step is the translation of λ-abstractions. We implemented two translation methods: λ-lifting [25] and SKBCI combinators [51]. For SKBCI, we omit the combinator definition axioms, because they are very explosive [10]. A third mode simply omits clauses containing λ-abstractions.

Evaluation and Discussion In Zipperposition, we can adjust the CPU time allotted to Ehoh, Ehoh's own parameters, the point when Ehoh is invoked, the number K_{size} of selected clauses from M_{ho}, and the λ translation method. We fix the time limit to 3 s, use Ehoh in *autoschedule* mode, and focus on the last three parameters. In *base*, collaboration with Ehoh is disabled (labeled −Ehoh).

Ehoh is invoked after $K_{\text{time}} \cdot t$ CPU seconds, where $0 \leq K_{\text{time}} < 1$ and t is the total CPU time allotted to Zipperposition. Figure 7 shows the effect of varying K_{time} when $K_{\text{size}} = 32$

	−Ehoh	16	32	64	128	256	512
TPTP	1635⋆	**1985**	1980	1978	1968	1968	1919
SH	452⋆	606	**608**	600	598	596	589

Fig. 9: Impact of the number of selected clauses K_{size}

and λ-lifting is used. The evaluation confirms that using a highly optimized backend such as Ehoh greatly improves the performance of a less optimized prover such as Zipperposition. The figure indicates that it is preferable to invoke the backend early. We have indeed observed that if the backend is invoked late, small clauses with deep derivations tend to be present by then. These clauses might have been used to delete important shallow clauses already. But due to their derivation depth, they will not be translated. In such situations, it is better to invoke the backend before the important clauses are deleted.

Figure 8 quantifies the effects of the three λ-abstraction translation methods. We fixed $K_{\text{time}} = 0.25$ and $K_{\text{size}} = 32$. The clear winner is λ-lifting. SKBCI combinators perform slightly better than omitting clauses containing λ-abstractions.

Figure 9 shows the effect of K_{size} on performance, with $K_{\text{time}} = 0.25$ and λ-lifting. Including a small number of higher-order clauses with the lowest weight performs better than including a large number of such clauses.

9 Comparison with Other Provers

Different choices of parameters lead to noticeably different sets of proved problems. In an attempt to use Zipperposition 2 to its full potential, we have created a portfolio mode that runs up to 50 configurations in parallel during the allotted time. The portfolio was designed to solve as many problems as possible from the TPTP benchmark set. To provide some context, we compare Zipperposition 2 with the latest versions of all higher-order provers that competed at CASC-J10: CVC4 1.9 [4], Leo-III 1.5.6 [44], Satallax 3.5 [11], and Vampire 4.5.1 [10]. The provers were run using the same parameters as in CASC, but with updated executables. Note that Vampire's higher-order schedule is optimized for running on a single core. We also include E 2.7 (more precisely, Ehoh, its higher-order configuration), the first version of this prover to syntactically support full higher-order logic, including λ-abstractions. Semantically, E 2.7 is arguably the weakest among the listed provers: It simply performs o-RW rewriting described in Sect. 3 followed by λ-lifitng before it applies λ-free superposition [7] (a precursor of all three higher-order superposition calculi) on the preprocessed problem.

We use the same benchmark sets as elsewhere in this article. To imitate the CASC-J10 setup, we use a 120 s wall-clock limit and a 960 s CPU limit. We even carried out our evaluation on the 8-core CPU nodes that were used for CASC-J10. We also ran Zipperposition in uncooperative mode, in which its collaboration with a backend is disabled. Figure 10 summarizes the results.

The evaluation results corroborate the CASC results. They also show that Zipperposition outperforms all other provers on SH benchmarks. This confirms our hypothesis that λ-superposition is a suitable basis for automatic higher-order reasoning. Further confirmation is provided by the success rate of Zipperposition's uncooperative version: Even without backend, Zipperposition is substantially better than all other provers on TPTP benchmarks, and it matches the performance of the top contenders on SH. On the other hand, the increase

	TPTP	SH
CVC4	1816	587
E	1980	676
Leo-III	2122	616
Satallax	2175	588
Vampire	2072	660
Zipperposition-uncoop	2311	652
Zipperposition	**2412**	**715**

Fig. 10: Comparison of competing higher-order theorem provers

in performance due to the addition of an efficient backend suggests that the implementation of this calculus in a modern first-order superposition prover such as E or Vampire would achieve even better results.

We believe that there are still techniques inspired by tableaux, SAT solving, and SMT solving that could be adapted and integrated in saturation provers. In particular, there are still 25 TPTP problems and 17 SH problems that can be proved by other provers but not by Zipperposition.

10 Discussion and Conclusion

Back in 1994, Kohlhase [28, Sect. 1.3] was optimistic about the future of higher-order automated reasoning:

> The obstacles to proof search intrinsic to higher-order logic may well be compensated by the greater expressive power of higher-order logic and by the existence of shorter proofs. Thus higher-order automated theorem proving will be practically as feasible as first-order theorem proving is now as soon as the technological backlog is made up.

For higher-order superposition, the backlog consisted of designing calculus extensions, heuristics, and algorithms that mitigate its weaknesses. In this article, we presented such enhancements, justified their design, and evaluated them. We explained how each weak point in the higher-order proving pipeline could be improved, from preprocessing to reasoning about formulas, to delaying unpromising or explosive inferences, to invoking a backend. Our evaluation indicates that higher-order superposition is now the state of the art in higher-order reasoning.

Higher-order extensions of first-order superposition have been considered by Bentkamp et al. [6, 7] and Bhayat and Reger [9, 10]. They introduced proof calculi, proved them refutationally complete, and suggested optional rules, but they hardly discussed the practical aspects of higher-order superposition. Extensions of SMT are discussed by Barbosa et al. [3]. Bachmair and Ganzinger [1], Manna and Waldinger [30], and Murray [32] have studied nonclausal resolution calculi.

In contrast, there is a vast literature on practical aspects of first-order reasoning using superposition and related calculi. The literature evaluates various procedures and techniques [22, 38], literal and term order selection functions [21], and clause evaluation functions [19, 41], among others. Our work joins the select club of papers devoted to practical aspects of higher-order reasoning [8, 16, 43, 58].

As a next step, we plan to implement the described techniques in E. We expect the resulting prover to be substantially more efficient than Zipperposition. Moreover, we want to investigate the proofs found by provers such as CVC4 and Satallax but missed by Zipperposition. Finding the reason behind why Zipperposition fails to prove specific problems will likely result in useful new techniques.

Acknowledgment We are grateful to the maintainers of StarExec for letting us use their service. Ahmed Bhayat and Giles Reger guided us through details of Vampire 4.5. Ahmed Bhayat, Michael Färber, Mathias Fleury, Predrag Janičić, Mark Summerfield, and the anonymous reviewers suggested content, textual, and typesetting improvements. We thank them all.

Vukmirović, Bentkamp, 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 and Nummelin's research has received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

References

1. Bachmair, L., Ganzinger, H.: Non-clausal resolution and superposition with selection and redundancy criteria. In: A. Voronkov (ed.) LPAR '92, *LNCS*, vol. 624, pp. 273–284. Springer (1992)
2. Backes, J., Brown, C.E.: Analytic tableaux for higher-order logic with choice. J. Autom. Reason. **47**(4), 451–479 (2011)
3. Barbosa, H., Reynolds, A., El Ouraoui, D., Tinelli, C., Barrett, C.W.: Extending SMT solvers to higher-order logic. In: P. Fontaine (ed.) CADE-27, *LNCS*, vol. 11716, pp. 35–54. Springer (2019)
4. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: G. Gopalakrishnan, S. Qadeer (eds.) CAV 2011, *LNCS*, vol. 6806, pp. 171–177. Springer (2011)
5. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for full higher-order logic. In: A. Platzer, G. Sutcliffe (eds.) CADE-28, *LNCS*, vol. 12699, pp. 396–412. Springer (2021)
6. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas. J. Autom. Reason. **65**(7), 893–940 (2021)
7. Bentkamp, A., Blanchette, J.C., Cruanes, S., Waldmann, U.: Superposition for lambda-free higher-order logic. In: D. Galmiche, S. Schulz, R. Sebastiani (eds.) IJCAR 2018, *LNCS*, vol. 10900, pp. 28–46. Springer (2018)
8. Benzmüller, C., Sorge, V., Jamnik, M., Kerber, M.: Can a higher-order and a first-order theorem prover cooperate? In: F. Baader, A. Voronkov (eds.) LPAR 2004, *LNCS*, vol. 3452, pp. 415–431. Springer (2004)
9. Bhayat, A., Reger, G.: Restricted combinatory unification. In: P. Fontaine (ed.) CADE-27, *LNCS*, vol. 11716, pp. 74–93. Springer (2019)
10. Bhayat, A., Reger, G.: A combinator-based superposition calculus for higher-order logic. In: N. Peltier, V. Sofronie-Stokkermans (eds.) IJCAR 2020, Part I, *LNCS*, vol. 12166, pp. 278–296. Springer (2020)
11. Brown, C.E.: Reducing higher-order theorem proving to a sequence of SAT problems. J. Autom. Reason. **51**(1), 57–77 (2013)
12. Cruanes, S.: Extending superposition with integer arithmetic, structural induction, and beyond. Ph.D. thesis, École polytechnique (2015)
13. Czajka, L., Kaliszyk, C.: Hammer for Coq: Automation for dependent type theory. J. Autom. Reason. **61**(1-4), 423–453 (2018)
14. Denzinger, J., Kronenburg, M., Schulz, S.: DISCOUNT—a distributed and learning equational prover. J. Autom. Reason. **18**(2), 189–198 (1997)
15. Ebner, G., Blanchette, J., Tourret, S.: Unifying splitting. In: A. Platzer, G. Sutcliffe (eds.) CADE-28, *LNCS*, vol. 12699, pp. 344–360. Springer (2021)
16. Färber, M., Brown, C.E.: Internal guidance for Satallax. In: N. Olivetti, A. Tiwari (eds.) IJCAR 2016, *LNCS*, vol. 9706, pp. 349–361. Springer (2016)
17. Filliâtre, J., Paskevich, A.: Why3—where programs meet provers. In: M. Felleisen, P. Gardner (eds.) ESOP 2013, *LNCS*, vol. 7792, pp. 125–128. Springer (2013)
18. Ganzinger, H., Stuber, J.: Superposition with equivalence reasoning and delayed clause normal form transformation. In: F. Baader (ed.) CADE-19, *LNCS*, vol. 2741, pp. 335–349. Springer (2003)

19. Gleiss, B., Suda, M.: Layered clause selection for theory reasoning (short paper). In: N. Peltier, V. Sofronie-Stokkermans (eds.) IJCAR 2020, Part I, *LNCS*, vol. 12166, pp. 402–409. Springer (2020)
20. Henkin, L.: Completeness in the theory of types. J. Symb. Log. **15**(2), 81–91 (1950)
21. Hoder, K., Reger, G., Suda, M., Voronkov, A.: Selecting the selection. In: N. Olivetti, A. Tiwari (eds.) IJCAR 2016, *LNCS*, vol. 9706, pp. 313–329. Springer (2016)
22. Hoder, K., Voronkov, A.: Comparing unification algorithms in first-order theorem proving. In: B. Mertsching, M. Hund, M.Z. Aziz (eds.) KI 2009, *LNCS*, vol. 5803, pp. 435–443. Springer (2009)
23. Huet, G.P.: A unification algorithm for typed lambda-calculus. Theor. Comput. Sci. **1**(1), 27–57 (1975)
24. Jensen, D.C., Pietrzykowski, T.: Mechanizing *omega*-order type theory through unification. Theor. Comput. Sci. **3**(2), 123–171 (1976)
25. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: J. Jouannaud (ed.) FPCA 1985, *LNCS*, vol. 201, pp. 190–203. Springer (1985)
26. Kaliszyk, C., Urban, J.: HOL(y)Hammer: Online ATP service for HOL Light. Math. Comput. Sci. **9**(1), 5–22 (2015)
27. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: J. Leech (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon (1970)
28. Kohlhase, M.: A mechanization of sorted higher-order logic based on the resolution principle. Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany (1994)
29. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: N. Sharygina, H. Veith (eds.) CAV 2013, *LNCS*, vol. 8044, pp. 1–35. Springer (2013)
30. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. In: B.G. Buchanan (ed.) IJCAI-79, pp. 542–551. William Kaufmann (1979)
31. McCune, W., Wos, L.: Otter—the CADE-13 competition incarnations. J. Autom. Reason. **18**(2), 211–220 (1997)
32. Murray, N.V.: Completely non-clausal theorem proving. Artif. Intell. **18**(1), 67–85 (1982)
33. Nipkow, T.: Functional unification of higher-order patterns. In: E. Best (ed.) LICS 1993, pp. 64–74. IEEE Computer Society (1993)
34. Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: J.A. Robinson, A. Voronkov (eds.) Handbook of Automated Reasoning, pp. 335–367. Elsevier and MIT Press (2001)
35. Nummelin, V., Bentkamp, A., Tourret, S., Vukmirović, P.: Superposition with first-class Booleans and inprocessing clausification. In: A. Platzer, G. Sutcliffe (eds.) CADE-28, LNCS. Springer (2021)
36. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
37. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: G. Sutcliffe, S. Schulz, E. Ternovska (eds.) IWIL-2010, *EPiC Series in Computing*, vol. 2, pp. 1–11. EasyChair (2010)
38. Reger, G., Suda, M., Voronkov, A.: Playing with AVATAR. In: A.P. Felty, A. Middeldorp (eds.) CADE-25, *LNCS*, vol. 9195, pp. 399–415. Springer (2015)
39. Schulz, S.: E—a brainiac theorem prover. AI Commun. **15**(2-3), 111–126 (2002)
40. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: P. Fontaine (ed.) CADE-27, *LNCS*, vol. 11716, pp. 495–507. Springer (2019)
41. Schulz, S., Möhrmann, M.: Performance of clause selection heuristics for saturation-based theorem proving. In: N. Olivetti, A. Tiwari (eds.) IJCAR 2016, *LNCS*, vol. 9706, pp. 330–345. Springer (2016)
42. Steen, A.: Extensional paramodulation for higher-order logic and its effective implementation Leo-III. Ph.D. thesis, Free University of Berlin, Dahlem, Germany (2018)
43. Steen, A., Benzmüller, C.: There is no best β-normalization strategy for higher-order reasoners. In: M. Davis, A. Fehnker, A. McIver, A. Voronkov (eds.) LPAR-20, *LNCS*, vol. 9450, pp. 329–339. Springer (2015)
44. Steen, A., Benzmüller, C.: Extensional higher-order paramodulation in leo-iii. J. Autom. Reason. **65**(6), 775–807 (2021)
45. Stump, A., Sutcliffe, G., Tinelli, C.: Starexec: A cross-community infrastructure for logic solving. In: S. Demri, D. Kapur, C. Weidenbach (eds.) IJCAR 2014, *LNCS*, vol. 8562, pp. 367–373. Springer (2014)
46. Sultana, N., Blanchette, J.C., Paulson, L.C.: LEO-II and Satallax on the Sledgehammer test bench. J. Appl. Log. **11**(1), 91–102 (2013)
47. Sutcliffe, G.: The CADE ATP System Competition—CASC. AI Magazine **37**(2), 99–101 (2016)
48. 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)
49. Sutcliffe, G.: The CADE-27 automated theorem proving system competition—CASC-27. AI Commun. **32**(5-6), 373–389 (2019)
50. Sutcliffe, G.: The 10th IJCAR automated theorem proving system competition—CASC-J10. AI Commun. pp. 1–15 (2021). DOI 10.3233/AIC-201566
51. Turner, D.A.: Another algorithm for bracket abstraction. J. Symb. Log. **44**(2), 267–270 (1979)

52. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: A. Biere, R. Bloem (eds.) CAV 2014, *LNCS*, vol. 8559, pp. 696–710. Springer (2014)
53. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making higher-order superposition work. In: A. Platzer, G. Sutcliffe (eds.) CADE-28, *LNCS*, vol. 12699, pp. 415–432. Springer (2021)
54. Vukmirović, P., Blanchette, J.C., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic. In: T. Vojnar, L. Zhang (eds.) TACAS 2019, Part I, *LNCS*, vol. 11427, pp. 192–210. Springer (2019)
55. Vukmirović, P., Nummelin, V.: Boolean reasoning in a higher-order superposition prover. In: P. Fontaine, K. Korovin, I.S. Kotsireas, P. Rümmer, S. Tourret (eds.) PAAR-2020, *CEUR Workshop Proceedings*, vol. 2752, pp. 148–166. CEUR-WS.org (2020)
56. Vukmirović, P., Bentkamp, A., Nummelin, V.: Efficient full higher-order unification. In: Z.M. Ariola (ed.) FSCD, *LIPIcs*, vol. 167, pp. 5:1–5:17. Schloss Dagstuhl—Leibniz-Zentrum für Informatik (2020)
57. Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving. In: N. Peltier, V. Sofronie-Stokkermans (eds.) IJCAR 2020, Part I, *LNCS*, vol. 12166, pp. 316–334. Springer (2020)
58. Wisniewski, M., Steen, A., Kern, K., Benzmüller, C.: Effective normalization techniques for HOL. In: N. Olivetti, A. Tiwari (eds.) IJCAR 2016, *LNCS*, vol. 9706, pp. 362–370. Springer (2016)