New Techniques for Higher-Order Superposition

Petar Vukmirović¹, Alexander Bentkamp¹, Jasmin Blanchette¹,²,³, Simon Cruanes⁴, Visa Nummelin¹, and Sophie Tourret²

¹ Vrije Universiteit Amsterdam, Amsterdam, the Netherlands {p.vukmirovic,a.bentkamp,j.c.blanchette,visa.nummelin}@vu.nl
² Max-Planck-Institut für Informatik, Saarbrücken, Germany stourret@mpi-inf.mpg.de
³ Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
⁴ Aesthetic Integration, Austin, Texas, USA simon@aestheticintegration.com

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 native reasoning with formulas, and the need for new heuristics to curb the explosion of specific higher-order rules. We describe techniques that address this and their implementation in the Zipperposition theorem prover. Largely thanks to them, Zipperposition emerged as a leader in the higher-order division of the CASC-J10 competition.

1 Introduction

In the last decades, superposition-based first-order automatic theorem provers have emerged as useful reasoning tools. They dominate at the annual CASC [33] theorem prover competitions, winning first-order division since the competition’s inception. They are also used as backends to proof assistants [9, 18, 25], automatic higher-order theorem provers [30], and software verifiers [10]. The superposition calculus has only recently been extended to higher-order logic (simple type theory), resulting in λ-superposition [3], which we developed together with Waldmann, as well as SKBCI-superposition [6] by Bhayat and Reger. At the 2020 edition of the CASC theorem proving competition (CASC-J10), Zipperposition 2, a higher-order prover primarily based on λ-superposition, won the higher-order division, ending the long winning streak of the tableaux-based Satallax [8].

Although the core theory of λ-superposition has been discussed in detail, many aspects of its successful use in practice have not been described before. Furthermore, in higher-order logic, we have discovered that tableaux have some advantages over resolution and superposition. We show how some successful tableaux techniques can be simulated in a saturation prover, and how the complexity and explosiveness of λ-superposition can be kept under control in practice.

http://www.tptp.org/CASC/J10/
The main drawback of $\lambda$-superposition compared with SKBCI-superposition is that it includes inference rules with infinitely many conclusions. We describe a mechanism that interleaves obtaining conclusions for infinitely branching inferences with the standard saturation process (Sect. 3). It allows the prover to exhibit the same behavior as first-order superposition on purely first-order problems, smoothly scaling with increased number of higher-order clauses. We also propose some heuristics to curb the explosion induced by highly prolific $\lambda$-superposition rules (Sect. 4).

Tableaux techniques take a more holistic view at a higher-order problem, as they work at the level of formulas, rather than clauses. We show how classification can be tightly integrated with the saturation process, as well as how information can be extracted from input formulas to build useful heuristic (Section 5). Collaboration with external first-order provers is frequently used in higher-order theorem proving. Since $\lambda$-superposition gracefully generalizes standard superposition, waiting on a first-order backend to produce result can be counterproductive; yet Zipperposition is nowhere as efficient as E [27] or Vampire [20], so it still makes sense to integrate a backend. We describe how to achieve a balance between allowing the higher-order calculus to do native reasoning and delegating reasoning to a first-order backend (Sect. 6). Finally, interesting patterns can be observed in various higher-order encodings of problems. We show how we can exploit them to simplify problems and enhance axiom selection (Sect. 7).

The described techniques are implemented in Zipperposition 2. At CASC-J10, Zipperposition 2 solved 84% of the given higher-order problems, 20 percentage points ahead of the next best prover, Satallax 3.4. Here we compare Zipperposition 2 with other provers on all monomorphic higher-order TPTP benchmarks [34] to perform a more extensive evaluation (Section 8).

2 Background and Setting

Our presentation and evaluation assume simply typed higher-order logic. The techniques can easily be generalized to support polymorphism, and indeed the implementation in Zipperposition is polymorphic.

**Higher-Order Logic.** We define terms inductively as free variables $F, G, X, Y$, bound variables $x, y, z$, constants $f, g, a, b$, applications $st$, or $\lambda$-abstractions $\lambda x.s$, where $s$ and $t$ are terms. The syntactic distinction between free and bound variables gives rise to *loose bound variables* (e.g., $y$ in $\lambda x.ya$) [21]. We abbreviate iterated application $(st_1)\cdots t_n$ to $s\overline{t}_n$ and $\lambda$-abstraction $\lambda x_1.\cdots .\lambda x_n.s$ to $\lambda\overline{x}_n.s$. Every $\beta$-normal term $s$ can be written as $\lambda\overline{x}_m.h\overline{t}_n$, where $h$ is not an application; we call $h$ the *head* of the term. If $h$ is a variable and its return type is Boolean, we call $h$ a *predicate variable*. A literal $l$ is an equation $s \approx t$ or a disequation $s \not\approx t$. A clause is a multiset of literals, interpreted disjunctively. Nonequational (predicate) literals are encoded as equations with $\top$ and $\bot$, based on their sign: For example, $\text{even}(x)$ becomes $\text{even}(x) \approx \top$, and and $\neg\text{even}(x)$ becomes $\text{even}(x) \approx \bot$. 
Higher-Order Superposition. The (Boolean-free) $\lambda$-superposition calculus is a refutational complete inference system and redundancy criterion for Boolean-free extensional polymorphic clausal higher-order logic. Vukmirović and Nummelin [39] describe a pragmatic extension to higher-order logic with Booleans. The calculus’s inference rules make extensive use of complete sets of unifiers (CSUs). The CSU for $s$ and $t$, denoted by $\text{CSU}(s, t)$, is a set of unifiers such that for any unifier $\varrho$ of $s$ and $t$, there exists a $\sigma \in \text{CSU}(s, t)$ such that $\varrho = \sigma \circ \theta$ for some substitution $\theta$. Since CSUs can be infinite, core $\lambda$-superposition rules such as superposition and equality resolution are infinitely branching.

By contrast, the SKBCI-superposition calculus avoids CSU computation by using a form of first-order unification, but it enumerates higher-order terms using rules that instantiate applied variables with partially applied combinators from the complete combinator set $\{S, K, B, C, I\}$. This calculus is the basis of Vampire 4.5, which finished closely behind Satallax 3.4 and 3.5 at CASC-J10.

Another finitely branching calculus is Satallax’s SAT-guided tableaux calculus [1]. Satallax was the leading higher-order prover of the 2010s. Its simple and elegant tableaux avoid deep superposition-style rewriting inferences. However, our working hypothesis for the past six years has been that superposition, having proved its value for first-order logic with equality already in the 1990s, would likely provide a stronger basis than tableaux for higher-order reasoning. SAT solvers can be exploited using the AVATAR architecture [36], and useful tableaux techniques can be integrated as well.

Zipperposition. Zipperposition 2 [?] is a higher-order theorem prover based on a pragmatic extension of $\lambda$-superposition. It was conceived as a testbed for rapid experimenting with extensions of first-order superposition, but over time, it has assimilated most of the techniques and heuristics of the state-of-the-art superposition prover E [27]. Zipperposition 2 also implements SKBCI-superposition, and some of the techniques we describe also apply to it.

Several of our techniques are concerned with extending the given clause algorithm, the standard saturation algorithm, to support higher-order reasoning patterns. This algorithm partitions the proof state in a set $P$ of passive clauses (initially containing all clauses) and a set $A$ of active clauses (initially empty). At each iteration of the procedure, a clause $C$ (called given clause) from $P$ is moved to $A$, all inferences between $C$ and clauses in $A$ are performed, and the conclusions are added to $P$. When $C$ is moved from passive to active set we say $C$ is activated or processed.

Experimental Setup. To assess the practical value of our proposed techniques, we carried out some experiments with Zipperposition 2. The experiments use all 2606 monomorphic higher-order problems from TPTP [34] 7.2.0 as the benchmarks. We fixed a base configuration of Zipperposition parameters as a baseline for all comparisons. Then, in each experiment, parameters that are concerned with a particular technique are varied to evaluate the technique. The experiments were performed on StarExec [31] servers, which are equipped with Intel Xeon E5-2609 0 CPUs clocked at 2.40 GHz. Unless otherwise stated, we used a
CPU time limit of 20 s, which is roughly the time each configuration is given in portfolio mode.

3 Enumerating Infinitely Branching Inferences

Integrating inference rules that require enumerating a CSU with the standard given clause algorithm is, both theoretically [40] and practically [29, Sect. 4.3.2], a relevant issue. Consider the higher-order resolution between clause $C \lor s \approx \top$ and $D \lor t \approx \bot$ which possibly yields infinitely many resolvents of the form $(C \lor D)\sigma$ where $\sigma \in \text{CSU}(s,t)$. A theorem prover that uses such a rule must implement a mechanism that fairly enumerates the resolvents between eligible clauses. We describe how to modify the given clause algorithm to enumerate the conclusions of inferences that rely on CSU elements.

To enumerate inference conclusions fairly, it is crucial to be able to evaluate a short prefix of the infinite sequence and save what remains of the sequence for later. However, since we cannot even decide whether there exists a next CSU element, the request for the next conclusion might not terminate, effectively bringing the theorem prover to a halt. Thus, we require that the procedure for CSU enumeration does not simply return a lazily evaluated stream [24, Sect. 4.2] of unifiers, but to also periodically return a distinguished back-off value $\ominus$, which signifies that control should be returned to the caller of the unification procedure. One such procedure is the complete unification procedure of Vukmirović et al. [38], but other unification procedures such as that of Huet [15] or Jensen and Pietrzykowski [16] can be modified to meet this requirement. Based on the stream of unifiers interspersed with $\ominus$, we can construct a stream of inferences (similarly interspersed with $\ominus$) for which finite prefixes of any size can be computed without worrying about nontermination.

To support infinite inference streams in the given clause algorithm, we extend it to represent the proof state not only by the active (A) and passive (P) clause sets, but also by the priority queue $Q$, which contains the inference streams. We have earlier described this extension very shortly (in a few sentences) [3], but here we describe its details together with heuristics used to postpone the unpromising streams.

As usual, all clauses are in $P$ initially. When a given clause $C$ is activated, all inference streams representing inferences between $C$ and clauses in $A$ are created. The streams are given a priority based on the kind of inference and the premises used, and they are stored in $Q$.

Conclusions of inferences in $Q$ must eventually be moved to $P$ for given clause algorithm to proceed. To this end, a stream $s$ with the highest priority is removed from $Q$ before the end of each iteration of the given clause algorithm. Then, the first element of $s$ is evaluated, yielding either a clause or $\ominus$. In the former case, the clause is simply returned. In the latter case, the remaining part of $s$, denoted $s'$, is probed for a clause up to $k$ times to avoid giving up on promising streams early (by default, $k = 20$). Finally, regardless of whether the clause is successfully obtained, $s'$ is stored in $Q$ with the updated priority. The new priority is based
on the number of times an ancestor of $s'$ was chosen for evaluation and whether a clause was obtained from $s$. Probing a single stream for a clause in each iteration of the given clause algorithm shifts the main choice point from choosing the given clause to choosing a stream in $Q$. As provers generally have better heuristics for choosing the given clause, in each iteration of the given clause algorithm we probe $n$ highest priority streams (by default, $n = 10$).

This process does not guarantee that each stream will eventually be visited. To ensure fairness, every $m$ (by default, $m = 70$) steps of the given clause algorithm *fair probing* is performed. Fair probing will probe $i$ oldest streams in $Q$ for a clause (as described above), where $i$ is initially 1 and increases with each fair probe. This achieves dovetailing. Finally, when the set of passive clauses is empty, we repeatedly visit all streams until $Q$ becomes empty, or until a stream yields a clause.

Although the described mechanism guarantees that each conclusion of rules with infinitely many conclusions will be considered, it can be improved. The unification procedure by Vukmirović et al. terminates on many fragments of higher-order terms including the first-order and pattern fragment [22]. If none of these simple fragment algorithms finds a unifier, the procedure immediately yields $\bot$ before continuing the search. Therefore, when given clause algorithm creates the inference streams, it will force computation of at most one conclusion (which is guaranteed to be efficient). If this results in a clause, the clause is put in the set $P$. The rest of the inference stream is stored in $Q$. This allows us to keep the same behavior as first-order resolution (or superposition) prover on first-order problems as conclusions with first-order most general unifiers will be immediately stored in the passive set.

The design of this mechanism was guided by folklore knowledge about higher-order theorem proving. First, it is our experience that most steps in long higher-order proofs are between first-order literals. The unification algorithm 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 penalty of the stream differently for each rule, allowing old streams of more useful inferences to be queried before newly added, but potentially less useful stream. Finally, if we use a unification algorithm that has aggressive redundancy elimination, we will often find the necessary unifier within the first few unifiers returned. Similarly, if a stream keeps returning $\bot$, it is likely that it is blocked in a nonterminating computation and should be ignored. Our heuristics to increase the stream penalties take into account both observations.

**Evaluation and Discussion.** Vukmirović et al. have implemented their unification procedure in Zipperposition, and integrated it with the described inference scheduling mechanism to enumerate conclusions of $\lambda$-superposition inference rules. Zipperposition is the only competing higher-order prover that solves all Church numeral problems from the TPTP library very efficiently [38]. On these hard unification problems, the stream system allows the prover to explore the proof state lazily. Consider the TPTP problem NUM800^1, which claims that there exists a function $F$ such that $F \, c_1 \, c_2 \approx c_2 \land F \, c_2 \, c_3 \approx c_6$, where $c_n$ stands for...
the $n$th Church numeral, $c_n = \lambda z.s^n(z)$, in which the exponent denotes iterated
application. To solve the problem, it suffices to take the multiplication operator
$\lambda x y z. x (y s) z$ as witness for $F$. However, this unifier is only one out of many
available for each occurrence of $F$. Provers that limit themselves to only finitely
many unifiers face a difficult heuristic choice: impose a shallow unification depth and
possibly miss the unifier, or deepen the unification and spend a long time
in the unification procedure. The time spent in the unification procedure can be
so long that no clauses get created. In an independent evaluation setup on the
same set of 2606 problems used in this paper, Vukmirović et al. [38, Sect. 7] com-
pared a complete variant and a pragmatic variant of the unification procedure,
both depth-limited. The pragmatic variant was used directly—all the inference
results were put in the passive set. As the complete version of the unification
procedure is much more explosive than the pragmatic version it was expected
that it would perform poorly. However, the evaluation showed that the complete
variant solved only 15 problems less than the most competitive pragmatic ver-
sion. Furthermore, it solved 19 problems not solved by the pragmatic version.
This shows that the stream system with its heuristics allows the prover to defer
exploring less promising branches of the unification and use the full potential of
complete higher-order unifier search to solve unification problems which cannot
be solved by impaired versions of the unification procedure.

Among the competing higher-order theorem provers, only Satallax uses in-
finently branching calculus rules. It maintains a queue of commands that contain
instructions how to create a successor state in the tableaux. One of the commands
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 execution of a command is guar-
anteed to make progress in enumerating the next closed functional term. Thus,
integration of infinitely branching rules which use CSU into the given clause
algorithm must be based on a more flexible system than Satallax’s commands.

4 Controlling Prolific Rules

To support higher-order features such as function extensionality and quantifica-
tion over functions, many refutationally complete calculi employ highly prolific
rules. For example, $\lambda$-superposition extends first-order superposition with a rule
$\text{FLUIDSUP} [3]$ that almost always applies to two clauses if one of them contains
a term of the form $F \pi_n$ ($n > 0$). We describe three mechanisms to keep rules
like these under control.

First, we penalize the streams of expensive inferences. The penalty of each
stream is given an initial value based on properties of the inference premises such
as their derivation depth. For prolific rules such as $\text{FLUIDSUP}$, we increment this
value by a parameter $p$. Penalties for less prolific variants of this rule, such as
$\text{DUPSUP}$, are increased by fraction of $p$ (e.g., $\lfloor p/3 \rfloor$).

Second, selection of prolific clauses is deferred. To select the given clause,
most saturating provers evaluate clauses using an evaluation function and se-
New Techniques for Higher-Order Superposition

Fig. 1: Effect of the priority function on the performance

<table>
<thead>
<tr>
<th>base</th>
<th>BAVN</th>
<th>PL</th>
<th>PSAV</th>
<th>PHOS</th>
<th>BNF</th>
<th>PDAV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1638</td>
<td><strong>1640</strong></td>
<td>1637</td>
<td>1632</td>
<td>1594</td>
<td>1520</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1: Effect of the priority function on the performance

lect the clause with the lowest evaluation. For this choice to be efficient, passive clauses are organized into a priority queue ordered by their evaluations. Inspired by E [27], Zipperposition maintains multiple queues (ordered by different evaluations) which are visited in a round-robin fashion; it also uses E’s two-layer evaluation functions, whose variation has recently been implemented in VAMPIRE [12].

Two layers of the clause evaluation 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 class. To control selection of prolific clauses, we introduce new clause priority functions that take into account features of a clause relevant for $\lambda$-superposition.

We implemented the priority function $\text{PreferHOSteps (PHOS)}$, which assigns a higher priority if $\lambda$-superposition-specific rules 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 results of higher-order inferences might be necessary to find the proof more efficiently.

Another new priority function we introduced is $\text{ByNormalizationFactor (BNF)}$, which is based on the observation that a higher-order inference that applies complicated substitution to a clause is usually followed by a $\beta\eta$-normalization step. If $\beta\eta$-normalization greatly reduced the size of a clause, it is likely that this substitution simplified the clause (e.g., by removing arguments of a variable). Thus, this function prefers clauses which are results of $\beta\eta$-normalization, and within those clauses prefers the ones with larger size reductions.

We have also implemented priority function $\text{PreferShallowAppVars (PSAV)}$ which prefers clauses with lower depths of the deepest occurrence of an applied variable. This function tries curbing the explosion of both $\lambda$- and SKBCI-superposition. Applying substitution to a top-level applied variable often reduces this applied variable to a term with a constant head, which likely results in a less explosive clause. We have also implemented functions $\text{PreferDeepAppVars (PDAV)}$, which returns the priority opposite of PSAV and $\text{ByAppVarNum (BAVN)}$ which prefers clauses with fewer occurrences of applied variables.

Last, we limit applicability of the prolific rules. We have observed that it often suffices to apply prolific higher-order rules only to initial clauses or clauses with shallow derivation depth. Thus, we implemented the option to forbid application of a rule if the derivation depth of any premise exceeds a limit.

**Evaluation and Discussion.** In the base configuration, Zipperposition visits several clause queues, one of which uses constant priority function. To evaluate different priority functions, we replaced the queue ordered by the constant priority with the queue ordered by priorities described above. We report the results in Figure 1, where columns denote different priority functions. To test if
different inference stream queue priorities make a difference on the success rate, we enabled FLUIDSUP and used different parameters $p$ for priority increment of FLUIDSUP inference queues. The rule FLUIDSUP is disabled in base. The results are shown in Figure 2. We have also tested limiting applicability of some rules to shallow clauses, but the results were too close to each other to make any interesting remark. In all of the figures in this paper, each cell gives the number of proved problems; the highest number is typeset in \textbf{boldface}.

Figure 1 shows that priority functions that require inspecting the proof of clauses (like PHOS and BNF) do not compensate their computational complexity: They decrease the overall success rate and together solve 11 problems that cannot be solved by base. Simple functions like PL are more effective: it solves 22 problems that base cannot solve, without decreasing the number of solved problems.

As expected, giving low penalty to FLUIDSUP is detrimental to performance. However, as the column for $p = 16$ in Figure 2 shows, we should not give too high a penalty increase either, since that delays useful FLUIDSUP inferences too much. Interestingly, even though configuration with $p = 1$ solves the least problems overall, it solves 7 problems not solved by base, which is more than any other configuration in the same figure.

5 Native Reasoning with Formulas

Higher-order logic identifies terms and formulas. In many cases, we can easily solve a problem by instantiating a predicate variable with the right formula. Finding this formula is usually easier if the problem is not clausified. Consider the problem $\exists F. F \ p q \leftrightarrow p \land q$. Expressed in this form, the problem is easily solvable by taking $F := \lambda xy.x \land y$. By contrast, the negated and clausified form $F \ p q \approx \bot \lor p \approx \top, F \ p q \approx \bot \lor q \approx \top, F \ p q \approx \top \lor q \approx \bot$. The clausification loses structural information which can be used as a heuristic for finding the correct instantiation. One of the strengths of tableau-based provers is that they do not clausify the input problem. This might explain Satallax’s dominance in the THF division of CASC competitions until CASC-J10.

Traditionally, superposition provers clausify the input problem, simplify the initial clause set, and start the saturation loop. Instead, we propose to integrate clausification tightly in the saturation loop. Instead of clausifying the initial problem, we represent an input formula $f$ as a higher-order clause $f \approx \top$. Vukmirovic and Nummelin extend $\lambda$-superposition with \textit{lazy clausification} rules [39, Sect. 3.4]. These incrementally clausify top-level logical symbols; for example, a clause $C' \lor (p \land q) \approx \bot$ yields a new clause $C' \lor p \approx \bot \lor q \approx \bot$.

Lazy clausification rules can be used as inference rules, which add conclusions to the passive set, or as simplification rules, which replace premises with conclu-
Inferences give more flexibility, since all intermediate clasification states will be stored in the proof state, at the cost of producing many clauses. Simplications produce fewer clauses while also allowing the prover to analyze the syntactic structure of formulas. Given that clausifying equivalences can destroy a lot of syntactic structure [11], we clausify equivalences only using inferences.

Integrating clausification with the proving process has many merits. First, during saturation, the prover might derive unit clauses, which can be used to demodulate any subterm in the unclausified part of the problem.

Second, if lazy clausification is used as an inference, λ-superposition inferences can be performed between formulas rather than clauses, potentially shortening proofs [39]. This is especially useful for the problem discussed above, which can be solved by negating the conjecture, eliminating the universal quantifier, and applying equality resolution to the literal $F p q \not\approx p \land q$.

Third, some superposition provers support variants of AVATAR [36], an architecture that makes it possible to split clauses into variable-disjoint subclauses and continue the proof search in fragments of the search space. Combining AVATAR and lazy clausification allows us to split a clause $(f_1 \lor \cdots \lor f_n) \approx \top$, where for all $i \neq j$, $f_i$ and $f_j$ do not share free variables, into variable disjoint clauses $f_i \approx \top$. When AVATAR is used in the first-order setting, $f_i$ cannot be formulas.

To finish the proof, it suffices to derive $\bot$ under each of the assumptions $f_i \approx \top$. As the split is performed on the formula level, this technique resembles tableaux, but it allows us to use the strengths of superposition, such as its powerful redundancy criterion and simplification machinery, to close the branches.

Finally, even when lazy clausification is used as a simplification rule, the prover can use the information obtained from the formulas on which clausification rules are applied to make heuristic choices. In particular, we look for λ-abstractions that return Booleans in each active clause and store them in a set $\text{Inst}$. Optionally, $\text{Inst}$ can contain primitive instantiations [39]—that is, imitations of logical symbols which approximate the shape of a formula that a predicate variable can be instantiated with.

Whenever lazy clausification replaces $x$ of function type in a clause of the form $(\forall x. f) \approx \top \lor C$ with a fresh variable $X$ to obtain $f(x \mapsto X) \approx \top \lor C$ we create additional clauses replacing $x$ with all terms $t \in \text{Inst}$ that are of the same type as $x$. However, as new term $t$ can be added to $\text{Inst}$ after a clause containing quantified variable of the same type as $t$ has been activated, we also save all clauses for which a universally bound function variable has been eliminated. When a term is added to $\text{Inst}$, we instantiate all suitable saved clauses. Instantiated clauses are not recognized as redundant as Zipperposition uses optimized, but incomplete subsumption algorithm.

By abstraction of $s_i$ (or $t_i$) in disequation $f \not\approx f$ we assume $\lambda x. u \approx v$, where $u$ and $v$ are obtained by replacing each occurrence of $s_i$ ($t_i$) with $x_i$. Analogous abstractions are defined for equations, with the signs appropriately flipped.

We have observed that adding abstractions of the literals in the conjecture to $\text{Inst}$ provides useful instantiations for formulas such as induction principles.
+A & −A \\
SC & 1624 & 1638 \\
LCI & 1496 & 1531 \\
LCS & 1659 & 1710 \\

Fig. 3: Interaction between the clausification method and basic AVATAR

for datatypes. Consider the problem DAT056ˆ2 [32] whose clausified conjecture is $ap\ x\ (ap\ y\ z) \not\approx \ ap\ (ap\ x\ y)\ z$, where $ap$ is the list append operator defined recursively on its first argument and $x, y$ and $z$ are lists. Abstracton of $x$ from the disequation yields $t = \lambda x.\ ap\ x\ (ap\ y\ z) \approx ap\ (ap\ x\ y)\ z$, which is inserted into Inst. One of the axioms in the problem is the induction axiom for the list datatype: $\forall\ p.\ (p\ nil\ \&\ (\forall\ x.\ p\ xs\ \rightarrow\ p\ (cons\ x\ xs))) \rightarrow\ \forall\ xs.\ p\ xs$, where nil and cons have the usual meanings. Together with equations defining $ap$, instantiating $p$ in this axiom with $t$ proves $\forall\ x.\ ap\ x\ (ap\ y\ z) \approx ap\ (ap\ x\ y)\ z$, which then easily leads to the proof of the conjecture.

When the problem is clausified the induction axiom is turned into several clauses. The proof is found when all of those clauses are instantiated with correct terms. When lazy clasification is used, this can be avoided by recognizing formulas suitable for instantiation during the proof search.

**Evaluation and Discussion.** The base configuration uses the standard clausification algorithm [23], both as a preprocessing step and after instantiating predicate variables. Zipperposition supports only a basic version of AVATAR [36], which is disabled in the base configuration. To test the merits of lazy clasification, we change base along two axes: We choose standard clasification (SC), lazy clasification as inference (LCI), or lazy clasification as simplification (LCS), and we either enable (+A) or disable (−A) basic AVATAR. The base configuration does not use Boolean instantiation.

As Figure 3 reveals, interleaving simplification techniques with clasification greatly increases the performance. On the other hand, using lazy clasification has the opposite effect. Overall, the basic version of AVATAR harms performance, but the sets of problems solved with and without AVATAR are vastly different. For example, SC+A configuration solves 60 problems not solved by SC−A.

Boolean instantiation is not compatible with SC. To test its effects, we selected the best configuration from Figure 3 (LCS without AVATAR), and enabled Boolean instantiation in this configuration. With this change, Zipperposition solves 1744 problems, with 36 unique solutions with respect to other configurations in Figure 3. Boolean instantiation is currently the only way Zipperposition 2 can solve some hard higher-order problems, such as DAT056ˆ2.

6 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 [29, Sect. 4.4], Satallax [8]
and Sledgehammer [7]. Those provers invoke first-order backends many times during a proof attempt and spend substantial time in backend collaboration. Since $\lambda$-superposition generalizes a highly efficient first-order calculus, we expect that future efficient $\lambda$-superposition implementations will not benefit much from backends. Experimental provers such as Zipperposition, however, can gain a lot. We describe some techniques for controlling the use of backends to make better use of $\lambda$-superposition calculus.

In his thesis [29, Sect. 6.1], Steen extensively evaluates effects of using different first-order backends on the performance of Leo-III. His results suggest that using several first-order backends is not worth the integration effort compared with using a single backend. Following these results, and since we want to limit the communication with backends, we focus our attention to using a single backend. Our backend of choice is Ehoh [37], an extension of E [27]. Ehoh is a state-of-the-art superposition prover with syntactic support for higher-order features such as partial application, applied variables, and formulas occurring as arguments of function symbols. Ehoh provides the efficiency of E, while making the translation from full higher-order logic easier: The only missing syntactic feature is $\lambda$-abstraction. On the other hand, Ehoh’s higher-order reasoning capabilities are limited as it is unable to synthesize $\lambda$-abstractions, essentially uses first-order unification, and does not perform any predicate variable instantiation.

In a departure from Leo-III and other cooperative provers, instead of regularly invoking the backend, we invoke it at most once per proof attempt. The reasoning behind this decision is that most competitive higher-order provers use a portfolio mode in which many configurations are run for a short time. Invoking a backend many times during a short run of a configuration would likely leave little time for native higher-order reasoning. Furthermore, superposition-based backends currently cannot be used incrementally, meaning that each invocation will start with no knowledge of the previous ones.

The choice of clauses to translate is critical for finding the proof. Let $I$ be the set of clauses representing the input problem; let $M$ denote the union of the active and passive set at the point of invoking the backend; last, let $M_{ho}$ denote the subset of $M$ that contains only clauses that were derived using at least one $\lambda$-superposition-specific inference rule. We order the clauses in $M_{ho}$ by the number of inferences used to derive the clause and by syntactic weight to break the ties. Then, we choose all clauses in $I$ and the first $n$ clauses from $M_{ho}$ for use with the backend reasoner. We include all the clauses in $I$ since they constitute the backbone of the initial problem. From all derived clauses in $M$, we include only the clauses from $M_{ho}$ because clauses derived using first-order rules will also be derived using Ehoh. We also expect large clauses with deep derivation to be less useful.

The remaining step is the translation of $\lambda$-abstractions. We support two translation methods: $\lambda$-lifting [17] and Curry’s SKBCI combinator translation [35] (without the corresponding combinator definition axioms, which are explosive [6]). Furthermore, a third mode simply avoids clauses containing $\lambda$-abstractions.
Evaluation and Discussion. Several parameters control how Zipperposition collaborates with its backend. These include the CPU time allotted for Ehoh, Ehoh’s own parameters, the moment of Ehoh’s invocation, the size of $M_{ho}$, and the $\lambda$-abstraction translation method. Due to limited resources, we fix the first two parameters to 5 seconds and automatic mode, respectively, and evaluate the last three parameters. In base, collaboration with E is disabled. Parameters are evaluated by enabling the collaboration with E and varying parameter values.

Ehoh is invoked after $p \cdot t$ CPU seconds, where $p \in [0, 1)$ and $t$ is the total CPU time allotted to Zipperposition. Figure 4 shows the effect of varying $p$ when the size of $M_{ho}$ is fixed to 32 and $\lambda$-lifting is used. The evaluation confirms that using a high-performance backend such as Ehoh greatly improves the performance of a less optimized prover such as Zipperposition. It seems preferable to invoke the backend early. By manual inspection we have observed that if the backend is invoked late, small clauses with deep derivation might be produced. Due to their derivation depth, they will not be translated, but they might make clauses with shallow derivation redundant. If the backend was invoked earlier, one of these shallow clauses could be translated and help produce the proof faster.

Figure 5 shows how the size $m$ of $M_{ho}$ affects the performance, using $p = 0.25$ and $\lambda$-lifting. Including a small number of higher-order clauses with the lowest weight performs better than including a large number of such clauses.

Figure 6 illustrates the effects of $\lambda$-abstraction translation methods, where “omitted” denotes that clauses containing $\lambda$-abstractions are not included in the translation. We fix $p = 0.25$ and $|M_{ho}| = 32$. Remarkably, removing clauses with $\lambda$-abstractions performs comparable to using the more sophisticated translations. Among the two translation methods, $\lambda$-lifting clearly wins.

7 Preprocessing Higher-Order Problems

The TPTP library contains thousands of higher-order problems. Even though these problems originate from various sources, they have a markedly different flavor from the first-order problems. Notably, they extensively use the definition annotation to identify universally quantified equations (or equivalences) that define symbols. For some problems, eagerly unfolding all the definitions and $\beta$-reducing eliminates all higher-order features, making the problem amenable
Fig. 6: Effect of $\lambda$-abstraction translation method on the performance to first-order methods. But often, unfolding the definitions inflates the problem beyond recognition. Thus, higher-order provers need to be flexible.

An effective way to deal with definitions is to interpret them as rewrite rules, using the orientation given in the input problem. If there are multiple definitions for the same symbol, only the first one is used as a rewrite rule. Then, whenever a clause is picked in the given clause procedure, it will be rewritten using the collected rules. Since the TPTP format enforces no constraints on definitions, this rewriting might not terminate. To ensure termination, we limit the number of applied rewrite steps. In practice, most TPTP problems are well behaved: Only one definition per symbol is given, and the definitions are not circular. Instead of rewriting a clause when it is activated, we can rewrite the input formulas as a preprocessing step. This has the advantage that input clauses will be fully simplified when the proving process starts, and no opaque defined symbols will occur in clauses, which often helps the heuristics.

Treating definitions as rewrite rules can be effective, but it compromises the refutational completeness of calculi such as $\lambda$-superposition and SKBCI-superposition, which depend on term orders to reduce the search space. More often than not, the order used is the Knuth–Bendix order (KBO) [19]. It compares terms by first comparing their weights, as given by a symbol weight assignment. It is easy to update a symbol weight assignment $W$ so that it orients acyclic definitions from left to right assuming they are of the form $f \overline{X}_m \approx \lambda \overline{y}_n. t$, where the only free variables in $t$ are $\overline{X}_m$, no free variable repeats 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 has no dependency on any other defined one. For each $f$, we set $W(f)$ to $w + 1$, where $w$ is the maximum weight of the right-hand sides of $f$’s definitions, computed using $W$. By construction, each left-hand side is heavier than the corresponding right-hand side, thereby justifying a left-to-right orientation.

To filter out axioms that are unlikely to be useful in a proof attempt, many theorem provers rely on the SInE algorithm [14]. SInE starts with the set of symbols occurring in the conjecture and tries to find axioms that define the properties of these symbols. Then, it finds the definitions of newly found symbols until it reaches a fixpoint. Axioms annotated with definition ease this search because they record the dependency between a symbol and its characterization that SInE is designed to discover. To exploit this information, we modified SInE to optionally include the definitions of symbols in the conjecture, regardless of whether they are filtered out or not. Similarly, we implemented mode of SInE which selects only conjecture and axioms annotated with definition.
Evaluation and Discussion. Axioms annotated with definition are treated as rewrite rules in base, and it preprocesses the formulas using the rewrite rules. In Figure 7, we also test the effects of disabling this preprocessing (− RW preprocess), disabling the special treatment of definition axioms (−RW), and disabling the special treatment of definition while using adjusted KBO weight as described above ((−)RW+KBO). The results show that treating definition axioms as rewrite rules greatly improves the performance. Adjusting the KBO weights yields many unique solutions.

8 Prover Comparison

Results shown in the previous sections suggest that different choices of parameters lead to noticeably different sets of solved 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 sequence during the allotted run time. To provide some context, we compare Zipperposition 2 with the latest versions of all competing higher-order provers: CVC4 1.8 [2], Leo-III 1.5 [30], Satallax 3.5 [8], and Vampire 4.5 [20].

We use the same set of 2606 monomorphic higher-order TPTP 7.2.0 problems as elsewhere in this paper, but we try to replicate the setup of CASC more faithfully. CASC was run on 8-core CPUs with a 120 s wallclock limit and a 960 s CPU limit. As we run our experiments on four-core CPUs, we set the wallclock limit to 240 s and keep the same CPU limit. Leo-III, Satallax, and Zipperposition are cooperative provers that can use backend provers. We also run them in uncooperative mode to measure their intrinsic strength. Figure 8 summarizes the results.

Among the cooperative provers, Zipperposition is the one that depends the less on its backend, and its uncooperative mode is only 2 problems behind Satallax. This confirms our intuition that λ-superposition is a good basis for automatic higher-order reasoning. The increase 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. Moreover, we believe that there are still techniques inspired by tableaux, SAT solving, and SMT solving that could be adapted to saturation provers.

9 Discussion and Conclusion

The research presented in this paper had two goals. First, we suggested solutions to issues that arise when implementing $\lambda$-superposition and, to a lesser extent, SKBCI-superposition. Second, we discussed many choices that can be made during proof search and provided a detailed evaluation for each such choice.

Extensions of first-order superposition to higher-order logic have been described in detail from a theoretical perspective, by Bentkamp et al. [3, 4] and Bhayat and Reger [5, 6]. They describe calculi, prove them refutationally complete, and discuss some optional rules. However, none of these descriptions discusses practical aspects of higher-order superposition such as heuristics and curbing the explosion induced by highly prolific higher-order rules. In contrast, there is a vast literature on practical aspects of first-order reasoning using superposition and related calculi. The literature provides detailed evaluation of various algorithms and techniques [13, 26], literal selection functions [28], and clause evaluation functions [12], among others.

In this paper, we described some ways in which the explosion incurred by $\lambda$-superposition can be kept under control. We designed a saturation loop that enumerates infinite sets of higher-order inference conclusions, while maintaining the same behavior as the standard given clause procedure on first-order clauses. For many choices that can be made during the higher-order proof search, we described heuristics and evaluated them. We also showed how native support for a Boolean type can dramatically improve a higher-order prover’s performance. We implemented the techniques in Zipperposition 2, which now clearly outperforms the competition on TPTP benchmarks.

As a next step, we plan to implement the described techniques in Ehoh, the $\lambda$-free higher-order extension of 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 solve a problem will likely result in useful new techniques.

Acknowledgment. We are grateful to the maintainers of StarExec for letting us use their service. 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