Boolean Reasoning in a Higher-Order Superposition Prover

Petar Vukmirović and Visa Nummelin

Vrije Universiteit Amsterdam,
Amsterdam, The Netherlands

Abstract

We present a pragmatic approach to extending a Boolean-free higher-order superposition calculus to support Boolean reasoning. Our approach extends inference rules that have been used only in a first-order setting, uses some well-known rules previously implemented in higher-order provers, as well as new rules. We have implemented the approach in the Zipperposition theorem prover. The evaluation shows highly competitive performance of our approach and clear improvement over previous techniques.

1 Introduction

In the last decades, automatic theorem provers have been used successfully as backends to "hammers" in proof assistants [16, 24] and to software verifiers [14]. Most advanced provers, such as CVC4 [4], E [28], and Vampire [20], are based on first-order logic, whereas most frontends that use them are based on versions of higher-order logic. Thus, there is a large gap in expressiveness between front- and backends. This gap is bridged using well-known translations from higher-order to first-order logic [22, 26]. However, translations are usually less efficient than native support [3, 5, 35]. The distinguishing features of higher-order logic used by proof assistants that the translation must eliminate include λ-binders, function extensionality – the property that functions are equal if they agree on every argument, described by the axiom $\forall(x, y : \tau \to \nu).\ (\forall(z : \tau).\, x\, z \approx y\, z) \Rightarrow x \approx y$, and formulas occurring as arguments of function symbols [22].

A group of authors including Vukmirović [5] recently designed a complete calculus for extensional Boolean-free higher-order logic. This calculus is an extension of superposition, the calculus used in most successful provers such as E or Vampire. The extension removes the need to translate the first two above mentioned features of higher-order logic. Kotelnikov et al. [18,19] extended the language of first-order logic to support the third feature of higher-order logic that requires translation. They described two approaches: one based on calculus-level treatment of Booleans and the other, which requires no changes to the calculus, based on preprocessing.

To fully bridge the gap between higher-order and first-order tools, we combine the two approaches: we use the efficient higher-order superposition calculus and extend it with inference rules that reason with Boolean terms. In early work, Kotelnikov et al. [19] have described a *FOOL paramodulation* rule that, under some order requirements, removes the need for the axiom describing the Boolean domain – $\forall(p : o).\ p \approx \top \lor p \approx \bot$. In this approach, it is assumed that a problem with formulas occurring as arguments of symbols is translated to first-order logic.

The backbone of our approach is based on an extension of this rule to higher-order logic. Namely, we do not translate away any Boolean structure that is nested inside non-Boolean terms and allow our rule to hoist the nested Booleans to the literal level. Then, we clausify the resulting formula (i.e., a clause that contains formulas in literals) using a new rule.

An important feature that we inherit by building on top of Bentkamp et al. [5] is support for (function) extensionality. Moving to higher-order logic with Booleans also means that we need to consider *Boolean extensionality*: $\forall(p : o)(q : o).\,(p \Leftrightarrow q) \Rightarrow p \approx q$. We extend the rules of Bentkamp et al. that treat function extensionality to support Boolean extensionality as well.

Rules that extend the two orthogonal approaches form the basis of our support for Boolean reasoning (Section 3). In addition, we have implemented rules that are inspired by the ones used in the higher-order provers Leo-III [30] and Satallax [10], such as elimination of Leibniz equality, primitive instantiation and treatment of choice operator [1]. We have also designed new rules that use higher-order unification to resolve Boolean formulas that are hoisted to literal level, delay clausification of non-atomic literals, reason about formulas under λ-binders, and many others. Even though the rules we use are inspired by the ones of refutationally complete higher-order provers, we do not guarantee completeness of our extension of λ-superposition.

We compare our native approach with two alternatives based on preprocessing (Section 4). First, we compare it to an axiomatization of the theory of Booleans. Second, inspired by work of Kotelnikov et al. [18], we implemented the preprocessing approach that does not require introduction of Boolean axioms. We also discuss some examples, coming from TPTP [32], that illustrate advantages and disadvantages of our approach (Section 5).

Our approach is implemented in the Zipperposition theorem prover [12,13]. Zipperposition is an easily extensible open source prover that Bentkamp et al. used to implement their higher-order superposition calculus. We further extend their implementation.

We performed an extensive evaluation of our approach (Section 6). In addition to evaluating different configurations of our new rules, we have compared them to full higher-order provers CVC4, Leo-III, Satallax and Vampire. The results suggest that it is beneficial to natively support Boolean reasoning – the approach outperforms preprocessing-based approaches. Furthermore, it is very competitive with state-of-the-art higher order provers. We discuss the differences between our approach and the approaches we base on, as well as related approaches (Section 7).

2 Background

We base our work on Bentkamp et al.'s [5] extensional polymorphic clausal higher-order logic. We extend the syntax of this logic by adding logical connectives to the language of terms. The semantic of the logic is extended by interpreting Boolean type o as a two-element domain. This amounts to extending Bentkamp et al's fragment of higher-order logic to full-higher order logic (HOL). Our notation, definitions and the following text are largely based on Bentkamp et al.'s.

A signature is a quadruple $(\Sigma_{\mathsf{ty}}, \mathcal{V}_{\mathsf{ty}}, \Sigma, \mathcal{V})$ where Σ_{ty} is a set of type constructors, $\mathcal{V}_{\mathsf{ty}}$ is a set of type variables and Σ and $\mathcal{V}$ are sets of constants and term variables, respectively. We require nullary type constructors ι and o, as well as binary constructor $\rightarrow$ to be in Σ_{ty}. A type τ, υ is either a type variable $\alpha \in \mathcal{V}_{\mathsf{ty}}$ or of the form $\kappa(\tau_1, \ldots \tau_n)$ where κ is an n-ary type constructor. We write κ for $\kappa()$, $\tau \rightarrow \upsilon$ for $\rightarrow(\tau, \upsilon)$, and we abbreviate tuples $(a_1, \ldots, a_n)$ as $\overline{a}_n$ for $n \geq 0$. Similarly, we drop parentheses to shorten $\tau_1 \rightarrow (\cdots \rightarrow (\tau_{n-1} \rightarrow \tau_n)\cdots)$ into $\tau_1 \rightarrow \cdots \rightarrow \tau_n$. Each symbol in Σ is assigned a type declaration of the form $\Pi\overline{\alpha}_n.\,\tau$ where all variables occurring in τ are among $\overline{\alpha}_n$.

Function symbols $\mathsf{a}, \mathsf{b}, \mathsf{f}, \mathsf{g}, \ldots$ are elements of Σ; their type declarations are written as $\mathsf{f} : \Pi\overline{\alpha}_n.\,\tau$. Term variables from the set $\mathcal{V}$ are written $x, y, z \ldots$ and we denote their types as $x : \tau$. When the type is not important, we omit type declarations. We assume that symbols $\top, \bot, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ with their standard meanings and type declarations are elements of Σ. Furthermore, we assume that polymorphic symbols $\forall$ and $\exists$ with type declarations $\Pi\alpha.\,(\alpha \rightarrow o) \rightarrow o$ and $\approx : \Pi\alpha.\,\alpha \rightarrow \alpha \rightarrow o$ are in Σ, with their standard meanings. All these symbols are called *logical symbols*. We write binary logical symbols in infix notation.

Terms are defined inductively as follows. Variables $x : \tau$ are terms of type τ. If $\mathsf{f} : \Pi\overline{\alpha}_n.\, \tau$ is in Σ and $\overline{\upsilon}_n$ is a tuple of types, called type arguments, then $\mathsf{f}\langle\overline{\upsilon}_n\rangle$ (written as f if $n = 0$, or if type arguments can be inferred from the context) is a term of type $\tau\{\overline{\alpha}_n \mapsto \overline{\upsilon}_n\}$, called constant. If x is a variable of type τ and s is a term of type υ then $\lambda x.\, s$ is a term of type $\tau \to \upsilon$. If s and t are of type $\tau \to \upsilon$ and τ, respectively, then $s\, t$ is a term of type υ. We call terms of Boolean type (o) *formulas* and denote them by $f, g, h, \ldots$; we use $p, q, r, \ldots$ for variables whose result type is o and $\mathsf{p}, \mathsf{q}, \mathsf{r}$ for constants with the same result type. We shorten iterated lambda abstraction $\lambda x_1.\, \ldots\, \lambda x_n.\, s$ to $\lambda \overline{x}_n.\, s$, and iterated application $(s\, t_1) \cdots t_n$ to $s\, \overline{t}_n$. We assume the standard notion of free and bound variables, capture-avoiding substitutions $\sigma, \rho, \theta, \ldots$, and α-, β-, η-conversion. Unless stated otherwise, we view terms as $\alpha\beta\eta$-equivalence classes, with η-long β-reduced form as the representative. Each term s can be uniquely written as $\lambda\overline{x}_m.\, a\, \overline{t}_n$ where a is either variable or constant and $m, n \geq 0$; we call a the *head* of s. We say that a term $a\, \overline{t}_n$ is written in *spine notation* [11]. Following our previous work [34], we define nonstandard notion of subterms and positions inductively as a graceful extension of the first-order counterparts: a term s is a subterm of itself at position ε. If s is a subterm of t_i at position p then s is a subterm of $a\, \overline{t}_n$ at position $i.p$, where a is a head. If s is a subterm of t at position p then s is a subterm of $\lambda x.\, t$ at position $1.p$. We use $s|_p$ to denote subterm of s at position p.

Given a formula f we call its Boolean subterm $f|_p$ a *top-level Boolean* if for all proper prefixes q of p, the head of $f|_q$ is a logical constant. Otherwise, we call it a *nested Boolean*. For example, in the formula $f = \mathsf{h}\, \mathsf{a} \approx \mathsf{g}\, (\mathsf{p} \Rightarrow \mathsf{q}) \vee \neg\mathsf{p}$, $f|_1$ and $f|_2$ are top-level Booleans, whereas $f|_{1.2.1}$ is a nested Boolean (as well as its subterms). Only top-level Booleans are allowed in first-order logic, whereas nested Booleans are characteristic for higher-order logic. A formula is called an *atom* if it is of the form $a\, \overline{t}_n$, where a is a non-logical head, or of the form $s \approx t$, where if s or t are of type o, and one of them has a logical head, the other one must be $\top$ or $\bot$. A *literal* L is an atom or its negation. A *clause* C is a multiset of literals, interpreted and written (abusing $\vee$) disjunctively as $L_1 \vee \cdots \vee L_n$. We write $s \not\approx t$ for $\neg(s \approx t)$. We say a variable is *free* in a clause C if it is not bound inside any subterm of a literal in C.

3 The Native Approach

Some support for Booleans was already present in Zipperposition before we started extending the calculus of Bentkamp et al. In this section, we start by describing the internals of Zipperposition responsible for reasoning with Booleans. We continue by describing 15 rules that we have implemented. For ease of presentation we divide them in three categories. We assume some familiarity with the superposition calculus [2] and adopt the notation used by Schulz [27].

3.1 Support for Booleans in Zipperposition

Zipperposition is an open source[1] prover written in OCaml. From its inception, it was designed as a prover that supports easy extension of its base superposition calculus to various theories, including arithmetic, induction and limited support for higher-order logic [12].

In Zipperposition, applications are represented in flattened, spine notation. In addition, Zipperposition uses associativity of $\wedge$ and $\vee$ to flatten out the nested applications of these symbols. For example, terms $\mathsf{p} \wedge (\mathsf{q} \wedge \mathsf{r})$ and $(\mathsf{p} \wedge \mathsf{q}) \wedge \mathsf{r}$ are internally stored as $\wedge\, \mathsf{p}\, \mathsf{q}\, \mathsf{r}$. Zipperposition's support for λ-terms is used to represent quantified nested Booleans: formulas $\forall x.\, f$ and $\exists x.\, f$ are represented as $\forall\, (\lambda x.\, f)$ and $\exists\, (\lambda x.\, f)$. After clausification of the input problem, no nested Booleans will be modified or renamed using fresh predicate symbols.

[1] https://github.com/sneeuwballen/zipperposition

The version of Zipperposition preceding our modifications distinguished between equational and non-equational literals. Following E [28], we modified Zipperposition to represent all literals equationally: a non-equational literal f is stored as $f \approx \top$, whereas $\neg f$ is stored as $f \not\approx \top$. Equations of the form $f \approx \bot$ and $f \not\approx \bot$ are transformed into $f \not\approx \top$ and $f \approx \top$, respectively.

3.2 Core Rules

Kotelnikov et al. [19], to the best of our knowledge, pioneered the approach of extending a first-order superposition prover to support nested Booleans. They call effects of including the axiom $\forall(p : o).\ p \approx \top \vee p \approx \bot$ a "recipe for disaster". To combat the explosive behavior of the axiom, they imposed the following two requirements to the simplification order $\succ$ (which is a parameter to the superposition calculus): $\top \succ \bot$ and $\top$ and $\bot$ are two smallest ground terms with respect to $\succ$. If these two requirements are met, there is no self-paramodulation of the clause and only paramodulation possible is from literal $p \approx \top$ of the mentioned axiom into a Boolean subterm of another clause. Finally, Kotelnikov et al. replace the axiom with the inference rule *FOOL Paramodulation* (FP):

$$\frac{C[f]}{C[\top] \vee f \approx \bot}\,\text{FP}$$

where f is a nested non-variable Boolean subterm of clause C, different from $\top$ and $\bot$. In addition, they translate the initial problem containing nested Booleans to first-order logic without interpreted Booleans; thus, symbols $\top$ and $\bot$, and type o correspond to proxy symbols and types introduced during the translation.

We created two rules that are syntactically similar to FP but are adapted for higher-order logic with one key distinction – we do not perform any translation:

$$\frac{C[f]}{C[\bot] \vee f \approx \top}\,\textsc{Cases} \qquad \frac{C[f]}{C[\bot] \vee f \approx \top \quad C[\top] \vee f \not\approx \top}\,\textsc{CasesSimp}$$

The double line in the definition of CasesSimp denotes that the premise is replaced by conclusions; obviously, the prover that uses the rules should not include them both at the same time. In addition, since literals $f \approx \bot$ are represented as negative equations $f \not\approx \top$, which cannot be used to paramodulate from, we change the first requirement on the order to $\bot \succ \top$.

These two rules hoist Boolean subterms f to the literal level; therefore, some results of Cases and CasesSimp will have literals of the form $f \approx \top$ (or $f \not\approx \top$) where f is not an atom. This introduces the need for the rule called eager clausification (EC):

$$\frac{C}{D_1 \ \cdots \ D_m}\,\text{EC}$$

We say that a clause is *standard* if all of its literals are of the form $s \mathrel{\dot\approx} t$, where s and t are not Booleans or of the form $f \mathrel{\dot\approx} \top$, where the head of f is not a logical symbol and $\dot\approx$ denotes $\approx$ or $\not\approx$. The rule EC is applicable if clause $C = L_1 \vee \cdots \vee L_n$ is not standard. The resulting clauses $\overline{D}_m$ represent the result of clausification of the formula $\forall\overline{x}.\ L_1 \vee \cdots \vee L_n$ where $\overline{x}$ are all free variables of C. Using Boolean extensionality, Zipperposition's clausification algorithm treats Boolean equality as equivalence (i.e., it replaces $\approx\langle o\rangle$ with $\Leftrightarrow$).

An advantage of leaving nested Booleans unmodified is that the prover will be able to prove some problems containing them without using the prolific rules described above. For example, given two clauses $\mathsf{f}\,(\mathsf{p}\,x \Rightarrow \mathsf{p}\,y) \approx \mathsf{a}$ and $\mathsf{f}\,(\mathsf{p}\,\mathsf{a} \Rightarrow \mathsf{p}\,\mathsf{b}) \not\approx \mathsf{a}$, the empty clause can easily be derived without the above rules. A disadvantage of this approach is that the proving process will periodically be interrupted by expensive calls to the clausification algorithm.

If implemented naively, rules Cases and CasesSimp can result in many redundant clauses. Consider the following example: let $\mathsf{p} : o \to o$, $\mathsf{a} : o$ and consider a clause set containing $\mathsf{p}\,(\mathsf{p}\,(\mathsf{p}\,(\mathsf{p}\,\mathsf{a}))) \approx \top$. Then, the clause $C = \mathsf{a} \approx \top \vee \mathsf{p}\,\bot \approx \top$ can be derived in eight ways using the rules, depending on which nested Boolean subterm was chosen for the inference. In general, if a clause has a subterm occurrence of the form $\mathsf{p}^n\,\mathsf{a}$, where both p and a have result type o, the clause $\mathsf{a} \approx \top \vee \mathsf{p}\,\bot \approx \top$ can be derived in 2^{n-1} ways. To combat these issues we implemented pragmatic restrictions of the rule: only f which is the leftmost outermost (or innermost) eligible subterm will be considered. With this modification C can be derived in only one way. Furthermore, some intermediate conclusions of the rules will not be derived, pruning the search space.

The clausification algorithm by Nonnengart and Weidenbach [23] aggressively simplifies the input problem using well-known Boolean equivalences before clausifying it. For example, the formula $\mathsf{p} \wedge \top$ will be replaced by p. To simplify nested Booleans we implemented the rule

$$\frac{C[f\sigma]}{\overline{\overline{C[g\sigma]}}}\,\textsc{BoolSimp}$$

where $f \longrightarrow g \in E$ runs over fixed set of rewrite rules E, and σ is any substitution. In the current implementation of Zipperposition, E consists of the rules described by Nonnengart and Weidenbach [23, Section 3]. This set contains the rules describing how each logical symbol behaves when either of its argument is $\top$ or $\bot$: for example, it includes $\top \Rightarrow p \longrightarrow p$ and $p \Rightarrow \top \longrightarrow \top$. Leo-III implements a similar rule, called simp [29, Section 4.2.1.].

Our decision to represent negative atoms as negative equations was motivated by the need to alter Zipperposition's earlier behavior as little as possible. Namely, negative atoms were not used as literals that can be used to paramodulate from, and as such added to the laziness of the superposition calculus. However, it might be useful to consider unit clauses of the form $f \not\approx \top$ as $f \approx \bot$ to strengthen demodulation. To that end, we have introduced the following rule:

$$\frac{f \not\approx \top \qquad C[f\sigma]}{\overline{\overline{f \not\approx \top \qquad C[\bot]}}}\,\textsc{BoolDemod}$$

3.3 Higher-Order Considerations

To achieve refutational completeness of higher-order resolution and similar calculi it is necessary to instantiate variables with result type o, *predicate variables*, with arbitrary formulas [1, 29]. Fortunately, we can approximate the formulas using a complete set of logical symbols (e.g., $\neg$, $\forall$, and $\wedge$). Since such an approximation is not only necessary for completeness of some calculi, but very useful in practice, we implemented the *primitive instantiation* (PI) rule:

$$\frac{C \vee \lambda \overline{x}_m.\, p\,\overline{s}_n \mathrel{\dot{\approx}} t}{(C \vee \lambda \overline{x}_m.\, p\,\overline{s}_n \mathrel{\dot{\approx}} t)\{p \mapsto f\}}\,\text{PI}$$

where p is a free variable of the type $\tau_1 \to \cdots \to \tau_n \to o$. Choosing a different f that instantiates p, we can balance between explosiveness of approximating a complete set of logical symbols and incompleteness of pragmatic approaches. We borrow the notion of imitation from higher-order unification jargon [34]: we say that the term $\lambda \overline{x}_m.\, \mathsf{f}\,(y_1\,\overline{x}_m) \cdots (y_n\,\overline{x}_m)$ is an *imitation* of constant $\mathsf{f} : \tau_1 \to \cdots \to \tau_n \to \tau$ for some variable z of type $\nu_1 \to \cdots \to \nu_m \to \tau$. Variables $\overline{y}_n$ are fresh free variables, where each y_i has the type $\nu_1 \to \cdots \to \nu_m \to \tau_i$; variable x_i is of type ν_i.

Rule PI was already implemented by Simon Cruanes in Zipperposition, before we started our modifications. The rule has different modes that generate sets of possible terms f for $p : \tau_1 \to \cdots \to \tau_n \to o$: *Full*, *Pragmatic*, and $\mathit{Imit}_\star$ where $\star$ is an element of a set of logical constants $P = \{\wedge, \vee, \approx\langle\alpha\rangle, \neg, \forall, \exists\}$. Mode *Full* contains imitations (for p) of all elements of P. Mode *Pragmatic* contains imitations of $\neg$, $\top$ and $\bot$; if there exist indices i, j such that $i \neq j$ and $\tau_i = \tau_j$, it contains $\lambda\overline{x}_n.\, x_i \approx x_j$; if there exist indices i, j such that $i \neq j$, and $\tau_i = \tau_j = o$, then it contains $\lambda\overline{x}_n.\, x_i \wedge x_j$ and $\lambda\overline{x}_n.\, x_i \vee x_j$; if for some i, $\tau_i = o$, then it contains $\lambda\overline{x}_n.\, x_i$. Mode $\mathit{Imit}_\star$ contains imitations of $\top$, $\bot$ and $\star$ (except for $\mathit{Imit}_{\forall\exists}$ which contains imitations of both $\forall$ and $\exists$).

While experimenting with our implementation we have noticed some proof patterns that led us to come up with the following modifications. First, it often suffices to perform PI only on initial clauses – which is why we allow the rule to be applied only to the clauses created using at most k generating inferences. Second, if the rule was used in the proof, its premise is usually only used as part of that inference – which is why we implemented a version of PI that removes the clause after all possible PI inferences have been performed. We observed that the mode $\mathit{Imit}_\star$ is useful in practice since often only a single approximation of a logical symbol is necessary.

Efficiently treating axiom of choice is notoriously difficult for higher-order provers. Andrews formulates this axiom as $\forall(p : \alpha \to o).\, (\exists(x : \alpha).\, p\, x) \Rightarrow p\,(\varepsilon\, p)$, where $\varepsilon : \Pi\alpha.\ (\alpha \to o) \to \alpha$ denotes the *choice operator* [1]. After clausification, this axiom becomes $p\, x \not\approx \top \vee p\,(\varepsilon\, p) \approx \top$. Since term $p\, x$ matches any Boolean term in the proof state, this axiom is very explosive. Therefore, Leo-III [30] deals with the choice operator on the calculus level. Namely, whenever a clause $C = p\, x \not\approx \top \vee p\,(\mathsf{f}\, p) \approx \top$ is chosen for processing, C is removed from the proof state and f is added to set of choice functions CF (which initially contains just ε). Later, elements of CF will be used to heuristically instantiate the axiom of choice. We reused the method of recognizing choice functions, but generalized the rule for creating the instance of the axiom (assuming $\xi \in \mathit{CF}$):

$$\frac{C[\xi\, t]}{x\,(t\, y) \not\approx \top \vee x\,(t\,(\xi\,(\lambda z.\, x\,(t\, z)))) \approx \top}\ \textsc{Choice}$$

Let D be the conclusion of Choice. The fresh variable x in D acts as arbitrary context around t, the chosen instantiation for p from axiom of choice; the variable x can later be replaced by imitation of logical symbols to create more complex instantiations of the choice axiom. To generate useful instances early, we create $D\{x \mapsto \lambda z.\, z\}$ and $D\{x \mapsto \lambda z.\, \neg z\}$. Then, based on Zipperposition parameters, D will either be deleted or kept. Note that D will not subsume its instances, since the matching algorithm of Zipperposition is too weak for this.

Most provers natively support extensionality reasoning: Bhayat et al. [6] modify first-order unification to return unification constraints consisting of pairs of terms of functional type, whereas Steen relies on the unification rules of Leo-III's calculus [29, Section 4.3.3.] to deal with extensionality. Bentkamp et al [5] altered core generating inference rules of the superposition calculus to support extensionality. Instead of requiring that terms involved in the inference are unifiable, it is required that they can be decomposed into *disagreement pairs* such that at least one of the disagreement pairs is of functional type. Disagreement pairs of terms s and t of the same type are defined inductively using function dp: $\mathsf{dp}(s, t) = \emptyset$ if s and t are equal; $\mathsf{dp}(a\,\overline{s}_n, b\,\overline{t}_m) = \{(a\,\overline{s}_n, b\,\overline{t}_m)\}$ if a and b are different heads; $\mathsf{dp}(\lambda x.\, s, \lambda y.\, t) = \{(\lambda x.\, s, \lambda y.\, t)\}$; $\mathsf{dp}(a\,\overline{s}_n, a\,\overline{t}_n) = \bigcup_{i=1}^{n} \mathsf{dp}(s_i, t_i)$. Then the extensionality rules are stated as follows:

$$\frac{s \approx t \vee C \qquad u[s'] \mathrel{\dot{\approx}} v \vee D}{(s_1 \not\approx s'_1 \vee \cdots \vee s_n \not\approx s'_n \vee u[t] \mathrel{\dot{\approx}} v \vee C \vee D)\sigma}\ \textsc{ExtSup}$$

$$\frac{s \not\approx s' \lor C}{(s_1 \not\approx s'_1 \lor \cdots \lor s_n \not\approx s'_n \cdots \lor C)\sigma}\ \textsc{ExtER}$$

$$\frac{s \approx t \lor s' \approx u \lor C}{(s_1 \not\approx s'_1 \lor \cdots \lor s_n \not\approx s'_n \lor t \not\approx u \lor s' \approx u \lor C)\sigma}\ \textsc{ExtEF}$$

Rules ExtSup, ExtER, and ExtEF are extensional versions of superposition, equality resolution and equality factoring [27]. By Ext we denote the union of these three rules. In each of the rules, σ is a most general unifier of the types of s and s', and $\mathsf{dp}(s\sigma, s'\sigma) = \{(s_1, s'_1), \ldots, (s_n, s'_n)\}$. All side conditions for extensional rules are the same as for the standard rules, except that condition that s and s' are unifiable is replaced by the condition that at least one s_i is of functional type and that $n > 0$. This rule is easily extended to support Boolean extensionality by requiring that at least one s_i is of functional or type o, and adding the condition "$\mathsf{dp}(f, g) = \{(f, g)\}$ if f and g are different formulas" to the definition of dp.

Consider the clause $\mathsf{f}\,(\neg\mathsf{p} \lor \neg\mathsf{q}) \not\approx \mathsf{f}\,(\neg(\mathsf{p} \land \mathsf{q}))$. This problem is obviously unsatisfiable, since arguments of f on different sides of the disequation are extensionally equal; however, without Ext rules Zipperposition will rely on Cases(Simp) and EC rules to derive the empty clause. Rule ExtER will generate $C = \neg\mathsf{p} \lor \neg\mathsf{q} \not\approx \neg(\mathsf{p} \land \mathsf{q})$. Then, C will get clausified using EC, effectively reducing the problem to $\neg(\neg\mathsf{p} \lor \neg\mathsf{q} \Leftrightarrow \neg(\mathsf{p} \land \mathsf{q}))$, which is first-order.

Zipperposition restricts ExtSup by requiring that s and s' are not of function or Boolean types. If the terms are of function type, our experience is that better treatment of function extensionality is to apply fresh free variables (or Skolem terms, depending on the sign [5]) to both sides of a (dis)equation to reduce it to a first-order literal; Boolean extensionality is usually better supported by applying EC on the top-level Boolean term. Thus, for the following discussion we can assume s and s' are not λ-abstractions or formulas. Then, ExtSup is applicable if s and s' have the same head, and a functional or Boolean subterm. To speed up retrieval of such terms, we added an index that maps symbols to positions in clauses where they appear as a head of a term that has a functional or Boolean subterm. This index will be empty for first-order problems, incurring no overhead if extensionality reasoning is not needed. One more restriction we implemented is that we do not apply Ext rules if all disagreement pairs have at least one side whose head is a variable; those will be dealt with more efficiently using standard, non-extensional, versions of the rules. We also eagerly resolve literals $s_i \not\approx s'_i$ using at most one unifier returned by terminating, pragmatic variant of unification algorithm by Vukmirović et al. [34].

Expressiveness of higher-order logic allows users to define equality using a single axiom, called Leibniz equality [1]: $\forall(x : \alpha)(y : \alpha).\, (\forall(p : \alpha \to o).\, p\,x \Rightarrow p\,y) \Rightarrow x \approx y$. Leibniz equality often appears in TPTP problems. Since modern provers have the native support for equality, it is usually beneficial to recognize and replace occurrences of Leibniz equality.

Before we began our modifications, Zipperposition had a powerful rule that recognizes clauses that contain variations of Leibniz equality and instantiates them with native equality. This rule was designed by Simon Cruanes, and to the best of our knowledge, it has not been documented so far. With his permission we describe this rule as follows:

$$\frac{p\,\bar{s}^1_n \approx \top \lor \cdots \lor p\,\bar{s}^i_n \approx \top \lor p\,\bar{t}^1_n \not\approx \top \lor \cdots \lor p\,\bar{t}^j_n \not\approx \top \lor C}{(p\,\bar{s}^1_n \approx \top \lor \cdots \lor p\,\bar{s}^i_n \approx \top \lor C)\sigma}\ \textsc{ElimPredVar}$$

where p is a free variable, p does not occur in any s^l_k or t^l_k, or in C; σ is defined as $\{p \mapsto \lambda\bar{x}_n.\, \bigvee_{k=1}^{j}(\bigwedge_{l=1}^{n} x_l \approx t^k_l)\}$.

To better understand how this rule removes variable-headed negative literals, consider the clause $C = p\,\mathsf{a}_1\,\mathsf{a}_2 \approx \top \lor p\,\mathsf{b}_1\,\mathsf{b}_2 \not\approx \top \lor p\,\mathsf{c}_1\,\mathsf{c}_2 \not\approx \top$. Since all side conditions are fulfilled, the rule ElimPredVar will generate $\sigma = \{p \mapsto \lambda xy.\,(x \approx \mathsf{b}_1 \wedge y \approx \mathsf{b}_2) \vee (x \approx \mathsf{c}_1 \wedge y \approx \mathsf{c}_2)\}$. After applying σ to C and subsequent β-reduction, negative literal $p\,\mathsf{b}_1\,\mathsf{b}_2 \not\approx \top$ will reduce to $(\mathsf{b}_1 \approx \mathsf{b}_1 \wedge \mathsf{b}_2 \approx \mathsf{b}_2) \vee (\mathsf{b}_1 \approx \mathsf{c}_1 \wedge \mathsf{b}_2 \approx \mathsf{c}_2) \not\approx \top$, which is equivalent to $\bot$. Thus, we can remove this literal and all negative literals of the form $p\,\bar{t}_n \not\approx \top$ from C and apply σ to the remaining ones.

The previous rule removes all variables occurring in disequations in one attempt. We implemented two rules that behave more lazily, inspired by the ones present in Leo-III and Satallax:

$$\frac{p\,\bar{s}_n \approx \top \vee p\,\bar{t}_n \not\approx \top \vee C}{(s_i \approx t_i \vee C)\sigma}\,\textsc{ElimLeibniz}+ \qquad \frac{p\,\bar{s}_n \not\approx \top \vee p\,\bar{t}_n \approx \top \vee C}{(s_i \approx t_i \vee C)\sigma'}\,\textsc{ElimLeibniz}-$$

where p is a free variable, p does not occur in t_i, $\sigma = \{p \mapsto \lambda\bar{x}_n.\,x_i \approx t_i\}$ and $\sigma' = \{p \mapsto \lambda\bar{x}_n.\,\neg(x_i \approx t_i)\}$. This rule differs from ElimPredVar in three ways. First, it acts on occurrences of variables in both positive and negative literals. Second, due to its simplicity, it usually does not require EC as the following step. Third, it imposes much weaker conditions on p. However, removing all negative variables in one step might improve performance. Coming back to example of the clause $C = p\,\mathsf{a}_1\,\mathsf{a}_2 \approx \top \vee p\,\mathsf{b}_1\,\mathsf{b}_2 \not\approx \top \vee p\,\mathsf{c}_1\,\mathsf{c}_2 \not\approx \top$, we can apply ElimLeibniz+ using the substitution $\sigma = \{\lambda xy.\,x \approx \mathsf{b}_1\}$ to obtain the clause $C' = \mathsf{a}_1 \approx \mathsf{b}_1 \vee \mathsf{a}_1 \not\approx \mathsf{c}_1$.

3.4 Additional Rules

Zipperposition's unification algorithm [34] uses flattened representation of terms with logical operators $\wedge$ and $\vee$ for heads to unify terms that are not unifiable modulo $\alpha\beta\eta$-equivalence, but are unifiable modulo associativity and commutativity of $\wedge$ and $\vee$. Let $\diamond$ denote either $\wedge$ or $\vee$. When the unification algorithm is given two terms $\diamond\,\bar{s}_n$ and $\diamond\,\bar{t}_n$, where neither of $\bar{s}_n$ nor $\bar{t}_n$ contain duplicates, it performs the following steps: First, it removes all terms that appear in both $\bar{s}_n$ and $\bar{t}_n$ from the two argument tuples. Next, the remaining terms are sorted first by their head term and then their weight. Finally, an attempt is made to unify sorted lists pairwise. As an example, consider the problem of unifying the pair $\big(\wedge\,(\mathsf{p}\,\mathsf{a})\ (\mathsf{q}\,(\mathsf{f}\,\mathsf{a})),\ \wedge\,(\mathsf{q}\,(\mathsf{f}\,\mathsf{a}))\ (r\,(\mathsf{f}\,(\mathsf{f}\,\mathsf{a})))\big)$ where r is a free variable. If the arguments of $\wedge$ are simply sorted as described above, we would try to unify $\mathsf{p}\,\mathsf{a}$ with $\mathsf{q}\,(\mathsf{f}\,\mathsf{a})$, and fail to find a unifier. However, by removing term $\mathsf{q}\,(\mathsf{f}\,\mathsf{a})$ from the argument lists, we will be left with the problem $(\mathsf{p}\,\mathsf{a}, r\,(\mathsf{f}\,(\mathsf{f}\,\mathsf{a})))$ which has a unifier.

The winner of THF division of CASC-27 [33], Satallax [10], has one crucial advantage over Zipperposition: it is based on higher-order tableaux, and as such it does not require formulas to be converted to clauses. The advantage of tableaux is that once it instantiates a variable with a term, this instantiation naturally propagates through the whole formula. In Zipperposition, which is based on higher-order superposition, the original formula is clausified and instantiating a variable in a clause C does not automatically instantiate it in all clauses that are results of clausification of the same formula as C. To mitigate this issue, we have created extensions of equality resolution and equality factoring that take Boolean extensionality into account:

$$\frac{s \approx s' \vee C}{C\sigma}\,\textsc{BoolER} \qquad \frac{x\,\bar{s}_n \approx \top \vee s' \not\approx \top \vee C}{(x\,\bar{s}_n \approx \neg s' \vee C)\sigma}\,\textsc{BoolEF}{+}{-}$$

$$\frac{x\,\bar{s}_n \not\approx \top \vee s' \approx \top \vee C}{(x\,\bar{s}_n \approx \neg s' \vee C)\sigma}\,\textsc{BoolEF}{-}{+} \qquad \frac{x\,\bar{s}_n \not\approx \top \vee s' \not\approx \top \vee C}{(x\,\bar{s}_n \approx s' \vee C)\sigma}\,\textsc{BoolEF}{-}{-}$$

All side conditions except for the ones concerning the unifiability of terms are as in the original equality resolution and equality factoring rules. In rule BoolER, σ is a unifier of s and $\neg s'$. In the $+-$ and $-+$ versions of BoolEF, σ unifies $x\,\bar{s}_n$ and $\neg s'$, and in the remaining version it unifies $x\,\overline{s}_n$ and s'. Intuitively, these rules bring Boolean (dis)equations in the appropriate form for application of the corresponding base rules. It suffices to consider literals of the form $s \approx s'$ for BoolER since Zipperposition rewrites $s \Leftrightarrow t \approx \top$ and $\neg(s \Leftrightarrow t) \not\approx \top$ to $s \approx t$ (and does analogous rewriting into $s \not\approx t$).

Another approach to mitigate harmful effects of eager clausification is to delay it as long as possible. Following the approach by Ganzinger and Stuber [15], we represent every input formula f as a unit clause $f \approx \top$ and use the following lazy clausification (LC) rules:

$$\frac{(f \wedge g) \approx \top \vee C}{f \approx \top \vee C \quad g \approx \top \vee C}\,\mathrm{LC}_\wedge \qquad \frac{(f \vee g) \approx \top \vee C}{f \approx \top \vee g \approx \top \vee C}\,\mathrm{LC}_\vee \qquad \frac{(f \Rightarrow g) \approx \top \vee C}{f \not\approx \top \vee g \approx \top \vee C}\,\mathrm{LC}_\Rightarrow$$

$$\frac{(\neg f) \approx \top \vee C}{f \not\approx \top \vee C}\,\mathrm{LC}_\neg \qquad \frac{(\forall x.\, f) \approx \top \vee C}{f\{x \mapsto y\} \vee C}\,\mathrm{LC}_\forall \qquad \frac{(\exists x.\, f) \approx \top \vee C}{f\{x \mapsto \mathsf{sk}\langle\overline{\alpha}\rangle\,\overline{y}_n\} \vee C}\,\mathrm{LC}_\exists$$

$$\frac{f \approx g \vee C}{f \not\approx \top \vee g \approx \top \vee C \quad f \approx \top \vee g \not\approx \top \vee C}\,\mathrm{LC}_\approx$$

The rules described above are as given by Ganzinger and Stuber (adapted to our setting), with the omission of rules for negative literals ($f \not\approx \top$), which are easy to derive and which can be found in their work [15]. In $\mathrm{LC}_\approx$ we require both f and g to be formulas and at least one of them not to be $\top$. In $\mathrm{LC}_\forall$, y is a fresh variable, and in $\mathrm{LC}_\exists$, sk is a fresh symbol and $\overline{\alpha}$ and $\overline{y}_n$ are all the type and term variables occurring freely in $\exists x.\, f$.

Naive application of the LC rules can result in exponential blowup in problem size. To avoid this, we rename formulas that have repeated occurrences. We keep the count of all non-atomic formulas occurring as either side of a literal. Before applying the LC rule on a clause $f \dot{\approx} \top \vee C$, we check whether the number of f's occurrences exceeds the threshold k. If it does, based on the polarity of the literal $f \dot{\approx} \top$, we add the clause $\mathsf{p}\,\overline{y}_n \not\approx \top \vee f \approx \top$ (if the literal is positive) or $\mathsf{p}\,\overline{y}_n \approx \top \vee f \not\approx \top$ (if the literal is negative), where $\overline{y}_n$ are all free variables of f and p is a fresh symbol. Then, we replace the clause $f \dot{\approx} \top \vee C$ by $\mathsf{p}\,\overline{y}_n \dot{\approx} \top \vee C$.

Before the number of occurrences of f is checked, we first check (using a fast, incomplete matching algorithm) if there is a formula g, for which definition was already introduced, such that $g\sigma = f$, for some substitution σ. This check can have three outcomes. First, if the definition $\mathsf{q}\,\overline{x}_n$ is already introduced for g with the polarity matching that of $f \dot{\approx} \top$, then f is replaced by $(\mathsf{q}\,\overline{x}_n)\sigma$. Second, if the definition was introduced, but with different polarity, we create the clause defining g with the missing polarity, and replace f with $(\mathsf{q}\,\overline{x}_n)\sigma$. Last, if the there is no renamed formula g generalizing f, then we perform the previously described check.

In addition to reusing names for formula definitions, we reuse the Skolem symbols introduced by the $\mathrm{LC}_\exists$ rule. When $\mathrm{LC}_\exists$ is applied to $f = \exists x.\, f'$ we check if there is a Skolem $\mathsf{sk}\langle\overline{\alpha}_m\rangle\,\overline{y}_n$ introduced for a formula $g = \exists x.\, g'$, such that $g\sigma = f$. If so, the symbol sk is reused and $\exists x.\, f'$ is replaced by $f'\{x \mapsto (\mathsf{sk}\langle\overline{\alpha}_m\rangle\,\overline{y}_n)\sigma\}$. Renaming and name reusing techniques are inspired by the VCNF algorithm described by Reger et al. [25].

Rules Cases and CasesSimp deal with Boolean terms, but we need to rely on extensionality reasoning to deal with λ-abstractions whose body has type o. Using the observation that the formula $\forall \overline{x}_n.\, f$ implies that $\lambda \overline{x}_n.\, f$ is extensionally equal to $\lambda \overline{x}_n.\, \top$ (and similarly, if $\forall \overline{x}_n.\, \neg f$, then $\lambda \overline{x}_n.\, f \approx \lambda \overline{x}_n.\, \bot$), we designed the following rule (where all free variables of f are $\overline{x}_n$ and variables occurring freely in C):

$$\frac{C[\lambda \overline{x}_n.\, f]}{(\forall \overline{x}_n.\, f) \not\approx \top \vee C[\lambda \overline{x}_n.\, \top] \qquad (\forall \overline{x}_n.\, \neg f) \not\approx \top \vee C[\lambda \overline{x}_n.\, \bot]}\,\textsc{Interpret}\lambda$$

4 Alternative Approaches

An alternative to heavy modifications of the prover needed to support the rules described above is to treat Booleans as yet another theory. Since the theory of Booleans is finitely axiomatizable, simply stating those axioms instead of creating special rules might seem appealing. Another approach is to preprocess nested Booleans by hoisting them to the top level.

4.1 Axiomatization

A simple axiomatization of the theory of Booleans is given by Bentkamp et al. [5]. Following their approach, we introduce the proxy type *bool*, which corresponds to o, to the signature. We define proxy symbols t, f, not, and, or, impl, equiv, forall, exists, choice, and eq which correspond to the homologous logical constants from Section 2. In their type declarations o is replaced by *bool*.

To make this paper self-contained we include the axioms from Bentkamp et al. [5]. Definitions of symbols are computational in nature: symbols are characterized by their behavior on t and f. This also reduces interferences between different axioms. Axioms are listed as follows:

$$\begin{array}{ccc}
\mathsf{t} \not\approx \mathsf{f} & \mathsf{or}\,\mathsf{t}\,x \approx \mathsf{t} & \mathsf{equiv}\,x\,y \approx \mathsf{and}\,(\mathsf{impl}\,x\,y)\,(\mathsf{impl}\,y\,x) \\
x \approx \mathsf{t} \lor x \approx \mathsf{f} & \mathsf{or}\,\mathsf{f}\,x \approx x & \mathsf{forall}\langle\alpha\rangle\,(\lambda x.\,\mathsf{t}) \approx \mathsf{t} \\
\mathsf{not}\,\mathsf{t} \approx \mathsf{f} & \mathsf{impl}\,\mathsf{t}\,x \approx x & y \approx (\lambda x.\,\mathsf{t}) \lor \mathsf{forall}\langle\alpha\rangle\,y \approx \mathsf{f} \\
\mathsf{not}\,\mathsf{f} \approx \mathsf{t} & \mathsf{impl}\,\mathsf{f}\,x \approx \mathsf{t} & \mathsf{exists}\langle\alpha\rangle\,y \approx \mathsf{not}\,(\mathsf{forall}\langle\alpha\rangle\,(\lambda x.\,\mathsf{not}\,(y\,x))) \\
\mathsf{and}\,\mathsf{t}\,x \approx x & x \not\approx y \lor \mathsf{eq}\langle\alpha\rangle\,x\,y \approx \mathsf{t} & y\,x \approx \mathsf{f} \lor y\,(\mathsf{choice}\langle\alpha\rangle\,y) \approx \mathsf{t} \\
\mathsf{and}\,\mathsf{f}\,x \approx \mathsf{f} & x \approx y \lor \mathsf{eq}\langle\alpha\rangle\,x\,y \approx \mathsf{f} &
\end{array}$$

4.2 Preprocessing Booleans

Kotelnikov et al. extended VCNF, Vampire's algorithm for clausification, to support nested Booleans [18]. Vukmirović et al. extended the clausification algorithm of Ehoh, the lambda-free higher-order version of E, to support nested Booleans inspired by VCNF extension [35, Section 8]. Zipperposition and Ehoh share the same clausification algorithm, enabling us to reuse the extension with one notable difference: unlike in Ehoh, not all nested Booleans different from variables, $\top$ and $\bot$ will be removed. Namely, Booleans that are below λ-abstraction and contain λ-bound variables will not be preprocessed. They cannot be easily hoisted to the level of an atom in which they appear, since this process might leak any variables bound in the context in which the nested Boolean appears. Similar preprocessing techniques are used in other higher-order provers [36].

5 Examples

The TPTP library contains thousands of higher-order benchmarks, many of them hand-crafted to point out subtle interferences of functional and Boolean properties of higher order logic. In this section we discuss some problems from the TPTP library that illustrate the advantages and disadvantages of our approach.

In the last five instances of the CASC theorem proving competition, the core calculus of the best performing higher-order prover was tableaux – a striking contrast from first-order part of the competition dominated by superposition-based provers. TPTP problem `SET557^1` might shed some light on why tableaux-based provers excel on higher-order problems.

This problem conjectures that there is no surjection from a set to its power set:

$$\neg(\exists(x : \iota \to \iota \to o).\, \forall(y : \iota \to o).\, \exists(z : \iota).\, x\, z \approx y)$$

After negating the conjecture and clausification this problem becomes $\mathsf{sk}_1\,(\mathsf{sk}_2\, y) \approx y$ where sk_1 and sk_2 are Skolem symbols. Then, we can use ArgCong rule [5] which applies fresh variable w to both sides of the equation, yielding clause $C = \mathsf{sk}_1\,(\mathsf{sk}_2\, y)\, w \approx y\, w$. Most superposition-based higher-order theorem provers (such as Leo-III, Vampire and Zipperposition) will split this clause into two clauses $C_1 = \mathsf{sk}_1\,(\mathsf{sk}_2\, y)\, w \not\approx \top \vee y\, w \approx \top$ and $C_2 = \mathsf{sk}_1\,(\mathsf{sk}_2\, y)\, w \approx \top \vee y\, w \not\approx \top$. This clausification step makes the problem considerably harder. Namely, the clause C instantiated with the substitution $\{y \mapsto \lambda x.\, \neg(\mathsf{sk}_1\, x\, x), w \mapsto \mathsf{sk}_2\,(\lambda x.\, \neg(\mathsf{sk}_1\, x\, x))\}$ yields the empty clause. However, if the original clause is split into two as described above, Zipperposition will rely on PI rule to instantiate y with imitation of $\neg$ and on equality factoring to further instantiate this approximation. These desired inferences need to be applied on both new clauses and represent only a fraction of inferences that can be done with C_1 and C_2, reducing the chance of successful proof attempt. Rule BoolER imitates the behavior of tableaux prover: it essentially rewrites the clause C into $\neg(\mathsf{sk}_1\,(\mathsf{sk}_2\, y)\, w) \not\approx y\, w$ which makes finding the necessary substitution easy and does not require a clausification step.

Combining rule (Bool)ER with lazy clausification is very fruitful as the problem `SYO033^1` illustrates. This problem also contains the single conjecture

$$\exists(x : (\iota \to o) \to o).\, \forall(y : \iota \to o).(x\, y \Leftrightarrow (\forall(z : \iota).\, y\, z))$$

The problem is easily solved if we instantiate variable x with the constant $\forall$. Moreover, the prover does not have to blindly guess this instantiation for x, but can obtain it by unifying $x\, y$ with $\forall\, y$ (which is the η-short form of $\forall(z : \iota).\, y\, z$). However, when the problem is clausified, all quantifiers are removed. Then, Zipperposition only finds the proof if appropriate instantiation mode of PI is used, and if both clauses resulting from clausifying the negated conjecture are appropriately instantiated. In contrast, lazy clausification will derive the clause $x\,(\mathsf{sk}\, x) \not\approx \forall\,(\mathsf{sk}\, x)$ from the negated conjecture in three steps. Then, equality resolution results in an empty clause, swiftly finishing the proof without any explosive inferences. This effect is even more pronounced on problems `SYO287^5` and `SYO288^5`, in which critical proof step is instantiation of a variable with imitation of $\vee$ and $\wedge$. In configurations that do not use lazy clausification and BoolER, Zipperposition times out in any reasonable time limit; with those two options it solves mentioned problems in less than 100 ms.

In some cases, it is better to preprocess the problem. For example, TPTP problem `SYO500^1.005` contains many nested Boolean terms:

$$\mathsf{f}_0\,(\mathsf{f}_1\,(\mathsf{f}_1\,(\mathsf{f}_1\,(\mathsf{f}_2\,(\mathsf{f}_3\,(\mathsf{f}_3\,(\mathsf{f}_3\,(\mathsf{f}_4\,\mathsf{a})))))))) \approx \mathsf{f}_0\,(\mathsf{f}_0\,(\mathsf{f}_0\,(\mathsf{f}_1\,(\mathsf{f}_2\,(\mathsf{f}_2\,(\mathsf{f}_2\,(\mathsf{f}_3\,(\mathsf{f}_4\,(\mathsf{f}_4\,(\mathsf{f}_4\,\mathsf{a}))))))))))$$

In this problem, all functions f_i are of type $o \to o$, and constant a is of type o. FOOL unfolding of nested Boolean terms will result in exponential blowup in the problem size. However, superposition-based theorem provers are well-equipped for this issue: their CNF algorithms use smart simplifications and formula renaming to mitigate these effects. Moreover, when the problem is preprocessed, the prover is aware of the problem size before the proving process starts and can adjust its heuristics properly. E, Zipperposition and Vampire, instructed to perform FOOL unfolding, solve the problem swiftly, using their default modes. However, if problem is not preprocessed, Zipperposition struggles to prove it using Cases(Simp) and due to the large number of (redundant) clauses it creates, succeeds only if specific heuristic choices are made.

	a	lo	li
b	*1646*	**1648**	1640
$\mathrm{b_c}$	1644	1645	1644

Figure 1: Effect of the CASES(SIMP) rule on success rate

6 Evaluation

We performed extensive evaluation to determine usefulness of our approach. As our benchmark set, we used all 2606 monomorphic theorems from the TPTP library, given in THF format. All of the experiments described in this section were performed on StarExec [31] servers with Intel Xeon E5-2609 0 CPUs clocked at 2.40 GHz. The evaluation is separated in two parts that answer different questions: How useful are the new rules? How does our approach compare with state-of-the-art higher-order provers?

6.1 Evaluation of the Rules

For this part of the evaluation, we fixed a single well-performing Zipperposition configuration called *base* (b). Since we are testing a single configuration, we used the CPU time limit of 15 s – roughly the time a single configuration is given in a portfolio mode. Configuration b uses the pragmatic variant pv^2_{1121} of the unification algorithm given by Vukmirović et al. [34]. It enables BOOLSIMP rule, EC rule, PI rule in *Pragmatic* mode with $k = 2$, ELIMLEIBNIZ and ELIMPREDVAR rules, BOOLER rule, and BOOLEF rules. To evaluate the usefulness of all rules we described above, we enable, disable or change the parameters of a single rule, while keeping all other parameters of b intact. In figures that contain sufficiently different configurations, cells are of the form $n(m)$ where n is the total number of proved problems by a particular configuration and m is the number of unique problems that a given configuration solved, compared to the other configurations in the same figure. Intersections of rows and columns denote corresponding combination of parameters. Result for the base configuration is written in *cursive*; the best result is written in **bold**.

First, we tested different parameters of CASES and CASESSIMP rules. In Figure 1 we report the results. The columns correspond to three possible options to choose subterm on which the inference is performed: **a** stands for any eligible subterm, **lo** and **li** stands for leftmost outermost and leftmost innermost subterms, respectively. The rows correspond to two different rules: **b** is the base configuration, which uses CASESSIMP, and $\mathbf{b_c}$ swaps this rule for CASES. Although the margin is slim, the results show it is usually preferable to select leftmost-outermost subterm.

Second, we evaluated all the modes of PI rule with 3 values for parameter k: 1, 2, and 8 (Figure 2). The columns denote, from left to right: disabling the PI rule, *Pragmatic* mode, *Full* mode, and $\mathit{Imit}_\star$ modes with appropriate logical symbols. The rows denote different values of k. The results show that different values for k have a modest effect on success rate. The raw data reveal that when we focus our attention to configurations with $k = 2$, mode *Full* can solve 10 problems no other mode (including disabling PI rule) can. Modes $\mathit{Imit}_\land$ and *Pragmatic* solve 2 problems, whereas $\mathit{Imit}_\lor$ solves one problem uniquely. This result suggests that, even though this is not evident from Figure 2, sets of problems solved solved by different modes somewhat differ.

Figure 3 gives results of evaluating rules that treat Leibniz equality on the calculus level: EL stands for ELIMLEIBNIZ, whereas EPV denotes ELIMPREDVAR; signs $-$ and $+$ denote that rule

	−PI	b_p	b_f	$b_\wedge$	$b_\vee$	$b_\approx$	$b_\neg$	$b_{\forall\exists}$
$k = 1$		**1648**	1628	1637	1634	1630	1641	1637
$k = 2$	1636	*1646*	1629	1636	1631	1627	1638	1634
$k = 8$		1643	1625	1633	1631	1623	1637	1635

Figure 2: Effect of PI rule on success rate

	−EL	+EL
−EPV	1584 (0)	1644 (0)
+EPV	1612 (0)	***1646 (0)***

Figure 3: Effect of Leibniz equality elimination rules

	−BEF	+BEF
−BER	1644 (2)	1643 (0)
+BER	1645 (0)	***1646 (0)***

Figure 4: Effect of BoolER and BoolEF rules

is removed from or added to configuration b, respectively. Disabling both rules severely lowers the success rate. The results suggest that including ElimLeibniz is beneficial to performance.

Similarly, Figure 4 discusses merits of including (+) or excluding (−) BoolER (BER) and BoolEF (BEF) rules. Our expectations were that inclusion of those two rules would make bigger impact on success rate. It turned out that, in practice, most of the effects of these rules could be achieved using a combination of the PI rule and basic superposition calculus rules.

Combining these two rules with lazy clausification is more useful: when the rule EC is replaced by the rule LC, the success rate increases (compared to 1646 problems solved by *b*) to 1660 problems. We also discovered that reasoning with choice is useful: when rule Choice is enabled, the success rate increases to 1653. We determined that including or excluding the conclusion *D* of Choice, after it is simplified, makes no difference. Counterintuitively, disabling BoolSimp rule results in 1640 problems, which is only 6 problems short of configuration *b*. Disabling Ext and Interpret-λ rules results in solving 25 and 31 problems less, respectively. Raw data show that in total, using configurations from Figure 1 to Figure 4, 1682 problems can be solved.

Last, we compare our approach to alternatives. Axiomatizing Booleans brings Zipperposition down to a grinding halt: only 1106 problems can be solved using this mode. On the other hand, preprocessing is fairly competitive: it solves only 8 problems less than the *b* configuration.

6.2 Comparison with Other Higher-Order Provers

We compared Zipperposition with all higher-order theorem provers that took part in THF division of CASC-27 [33]: CVC4 1.8 prerelease [4], Leo-III 1.4 [30], Satallax 3.4 [10], and Vampire-THF 4.4 [20]. In this part of the evaluation, Zipperposition is ran in portfolio mode that runs configurations in different time slices. We set the CPU time limit to 180 s, the time allotted to each prover at CASC-27.

Leo-III and Satallax are cooperative theorem provers – they periodically invoke first-order provers to finish the proof attempt. Leo-III uses CVC4, E and iProver [17] as backends, while Satallax uses Ehoh [35] as backend. Zipperposition can use Ehoh as backend as well. To test how successful each calculus is, we run the cooperative provers in two versions: *pure*, which disables backends, and *coop* which uses all supported backends.

In both pure and cooperative mode, Satallax comes out as the winner. Zipperposition comes

	CVC4	Leo-III	Satallax	Vampire	Zipperposition
pure	1806 (5)	1627 (0)	2067 (0)	1924 (7)	1980 (0)
coop	–	2085 (3)	**2214 (9)**	–	2190 (17)

Figure 5: Comparison with other higher-order provers

in close second, showing that our approach is a promising basis for further extensions. Leo-III uses SMT solver CVC4, which features native support for Booleans, as a backend. It is possible that the use of CVC4 is one of the reasons for massive improvement in success rate of cooperative configuration of Leo-III, compared with the pure version. Therefore, we conjecture that including support for SMT backends in Zipperposition might be beneficial.

7 Discussion and Related Work

Our work is primarily motivated by the goal of closing the gap between higher-order "hammer" or software verifier frontends and first-order backends. Considerable amount of research effort has gone into making the translations of higher-order logic as efficient as possible. Descriptions of hammers like HOLyHammer [16] and Sledgehammer [24] for Isabelle contain details of these translations. Software verifiers Boogie [21] and Why3 [9] use similar translations.

Established higher-order provers like Leo-III and Satallax perform very well on TPTP benchmarks; however, recent evaluations show that on Sledgehammer problems they are outperformed by translations to first-order logic [3, 5, 35]. Those two provers are built from the ground up as higher-order provers – treatment of exclusively higher-order issues such as extensionality or choice is built into them usually using explosive rules. Those explosive rules might contribute to their suboptimal performance on mostly first-order Sledgehammer problems.

In contrast, our approach is to start with a first-order prover and gradually extend it with higher-order features. The work performed in the context of Matryoshka project [8], in which both authors of this paper participate, resulted in adding support for λ-free higher-order logic with Booleans to E [35] and veriT [3], and adding support for Boolean-free higher-order logic to Zipperposition. Authors of many all state-of-the-art first-order provers have implemented some form of support for higher-order reasoning. This is true both for SMT solvers, witnessed by the recent extension of CVC4 and veriT [3], and for superposition provers, witnessed by the extension of Vampire [7]. All of those approaches were arguably more focused on functional aspects of higher-order logic, such as λ-binders and function extensionality, than on Boolean aspects such as Boolean subterms and Boolean extensionality. A notable exception is work by Kotelnikov et al. that introduced support for Boolean subterms to first-order Vampire [18, 19].

The main merit of our approach is that it combines two successful complementary approaches to support features of higher-order logic that have not been combined before in a modular way. It is based on a higher-order superposition calculus that incurs around 1% of overhead on first-order problems compared with classic superposition [5]. We conjecture that it is this efficient reasoning base on which the approach is based that contributes to its competitive performance.

8 Conclusion

We presented a pragmatic approach to support Booleans in a modern automatic prover for clausal higher-order logic. Our approach combines previous research efforts that extended first-order provers with complementary features of higher-order logic. It also proposes some solutions for the issues that emerge with this combination. The implementation shows clear improvement over previous techniques and competitive performance.

What our work misses is an overview of heuristics that can be used to curb the explosion incurred by some of the rules described in this paper. In future work, we plan to address this issue. Similarly, unlike Bentkamp et al. [5], we do not give any completeness guarantees for our extension. We plan to develop a refutationally complete calculus that supports Booleans around core rules such as Cases and LC in future work.

Acknowledgment We are grateful to the maintainers of StarExec for letting us use their service. We thank Alexander Bentkamp, Jasmin Blanchette and Simon Cruanes for many stimulating discussions and all the useful advice. Alexander Bentkamp suggested the rule BoolER. We are also thankful to Ahmed Bhayat, Predrag Janičić and the anonymous reviewers for suggesting many improvements to this paper. We thank Evgeny Kotelnikov for patiently explaining the details of FOOL, Alexander Steen for clarifying Leo-III's treatment of Booleans, and Giles Reger and Martin Suda for explaining the renaming mechanism implemented in VCNF. Vukmirović'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). Nummelin 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] Peter B. Andrews. Classical type theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, pages 965–1007. Elsevier and MIT Press, 2001.

[2] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[3] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark W. Barrett. Extending SMT solvers to higher-order logic. In Pascal Fontaine, editor, *CADE 2019*, volume 11716 of *LNCS*, pages 35–54. Springer, 2019.

[4] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV 2011*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[5] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. Superposition with lambdas, 2020. Submitted to a journal, http://matryoshka.gforge.inria.fr/pubs/lamsup_report.pdf.

[6] Ahmed Bhayat and Giles Reger. A combinator-based superposition calculus for higher-order logic. Accepted at IJCAR 2020, https://github.com/vprover/vampire_publications/blob/master/paper_drafts/comb_sup_report.pdf.

[7] Ahmed Bhayat and Giles Reger. Restricted combinatory unification. In Pascal Fontaine, editor, *CADE 2019*, volume 11716 of *LNCS*, pages 74–93. Springer, 2019.

[8] Jasmin Blanchette, Pascal Fontaine, Stephan Schulz, Sophie Tourret, and Uwe Waldmann. Stronger higher-order automation: A report on the ongoing matryoshka project. In Martin Suda and Sarah Winkler, editors, *EPTCS 311: Proceedings of the Second International Workshop on*

Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements - Natal, Brazil, August 26, 2019, Electronic Proceedings in Theoretical Computer Science, EPTCS, pages 11–18. EPTCS, 12 2019.

[9] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wroclaw, Poland, 2011.

[10] Chad E. Brown. Satallax: An automatic higher-order prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 111–117. Springer, 2012.

[11] Iliano Cervesato and Frank Pfenning. A linear spine calculus. *J. Log. Comput.*, 13(5):639–688, 2003.

[12] Simon Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond.* PhD thesis, École polytechnique, 2015.

[13] Simon Cruanes. Superposition with structural induction. In Clare Dixon and Marcelo Finger, editors, *FroCoS 2017*, volume 10483 of *LNCS*, pages 172–188. Springer, 2017.

[14] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[15] Harald Ganzinger and Jürgen Stuber. Superposition with equivalence reasoning and delayed clause normal form transformation. *Inf. Comput.*, 199(1-2):3–23, 2005.

[16] Cezary Kaliszyk and Josef Urban. Hol(y)hammer: Online ATP service for HOL light. *Mathematics in Computer Science*, 9(1):5–22, 2015.

[17] Konstantin Korovin. iprover - an instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR 2008*, volume 5195 of *LNCS*, pages 292–298. Springer, 2008.

[18] Evgenii Kotelnikov, Laura Kovács, Martin Suda, and Andrei Voronkov. A clausal normal form translation for FOOL. In Christoph Benzmüller, Geoff Sutcliffe, and Raúl Rojas, editors, *GCAI 2016*, volume 41 of *EPiC Series in Computing*, pages 53–71. EasyChair, 2016.

[19] Evgenii Kotelnikov, Laura Kovács, and Andrei Voronkov. A first class boolean sort in first-order theorem proving and TPTP. *CoRR*, abs/1505.01682, 2015.

[20] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV 2013*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

[21] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *TACAS 2010*, volume 6015 of *LNCS*, pages 312–327. Springer, 2010.

[22] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.

[23] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 335–367. Elsevier and MIT Press, 2001.

[24] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *IWIL-2010*, volume 2 of *EPiC*, pages 1–11. EasyChair, 2012.

[25] Giles Reger, Martin Suda, and Andrei Voronkov. New techniques in clausal form generation. In Christoph Benzmüller, Geoff Sutcliffe, and Raúl Rojas, editors, *GCAI 2016*, volume 41 of *EPiC Series in Computing*, pages 11–23. EasyChair, 2016.

[26] J.A. Robinson. A note on mechanizing higher order logic. In B. Meltzer and D. Michie, editors,

Machine Intelligence, volume 5, pages 121–135. Edinburgh University Press, 1970.

[27] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3):111–126, 2002.

[28] Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *CADE 2019*, volume 11716 of *LNCS*, pages 495–507. Springer, 2019.

[29] Alexander Steen. *Extensional paramodulation for higher-order logic and its effective implementation Leo-III*. PhD thesis, Free University of Berlin, Dahlem, Germany, 2018.

[30] Alexander Steen and Christoph Benzmüller. The higher-order prover Leo-III. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *IJCAR 2018*, volume 10900 of *LNCS*, pages 108–116. Springer, 2018.

[31] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *IJCAR 2014*, volume 8562 of *LNCS*, pages 367–373. Springer, 2014.

[32] Geoff Sutcliffe. The TPTP problem library and associated infrastructure - from CNF to TH0, TPTP v6.4.0. *J. Autom. Reasoning*, 59(4):483–502, 2017.

[33] Geoff Sutcliffe. The CADE-27 automated theorem proving system competition - CASC-27. *AI Commun.*, 32(5-6):373–389, 2019.

[34] Petar Vukmirović, Alexander Bentkamp, and Visa Nummelin. Efficient full higher-order unification. Technical report, 2020. Accepted at FSCD 2020, http://matryoshka.gforge.inria.fr/pubs/hounif_paper.pdf.

[35] Petar Vukmirovic, Jasmin Christian Blanchette, Simon Cruanes, and Stephan Schulz. Extending a brainiac prover to lambda-free higher-order logic, 2020. Submitted to a journal, http://matryoshka.gforge.inria.fr/pubs/ehoh_article.pdf.

[36] Max Wisniewski, Alexander Steen, Kim Kern, and Christoph Benzmüller. Effective normalization techniques for HOL. In Nicola Olivetti and Ashish Tiwari, editors, *IJCAR 2016*, volume 9706 of *LNCS*, pages 362–370. Springer, 2016.