SAT-Inspired Eliminations for Superposition

Petar Vukmirović¹, Jasmin Blanchette¹,², Marijn J.H. Heule³

¹Vrije Universiteit Amsterdam, Amsterdam, the Netherlands
²Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
³Carnegie Mellon University, Pittsburgh, Pennsylvania, United States

Abstract—Optimized SAT solvers not only preprocess the clause set, they also transform it during solving as inprocessing. Some preprocessing techniques have been generalized to first-order logic with equality. In this paper, we port inprocessing techniques to work with superposition, and we strengthen preprocessing. Specifically, we look into elimination of hidden literals, variables (predicates), and blocked clauses. Our evaluation using the Zipperposition prover confirms that the new techniques usefully supplement the existing superposition machinery.

I. INTRODUCTION

Automated reasoning tools have become much more powerful in the last few decades thanks to procedures such as conflict-driven clause learning (CDCL) [1] for propositional logic and superposition [2] for first-order logic with equality. However, the effectiveness of these procedures crucially depends on how the input problem is represented as a clause set. The clause set can be optimized beforehand (preprocessing) or during the execution of the procedure (inprocessing). In this paper, we lift several preprocessing and inprocessing techniques from propositional logic to clausal first-order logic and demonstrate their usefulness in a superposition prover.

For many years, SAT solvers have used inexpensive clause simplification techniques such as hidden literal and hidden tautology elimination [3], [4] and failed literal detection [5, Sect. 1.6]. We generalize these techniques to first-order logic with equality (Sect. III). Since the generalization involves reasoning about infinite sets of literals, we propose restrictions to make them usable.

Variable elimination, based on Davis–Putnam resolution [6], has been studied in the context of both propositional logic [7], [8] and quantified Boolean formulas (QBFs) [9]. The basic idea is to resolve all clauses with negative occurrences of a propositional variable (i.e., a nullary predicate symbol) against clauses with positive occurrences and delete the parent clauses. Eén and Biere [10] refined the technique to identify a subset of clauses that effectively define a variable and use it to further optimize the clause set. This latter technique, variable elimination by substitution, has been an important preprocessor component in many SAT solvers since its introduction in 2004.

Specializing second-order quantifier elimination [11], [12], Khasidashvili and Korovin [13] generalized variable elimination to preprocess first-order problems, yielding a technique we call singular predicate elimination. We extend their work along two axes (Sect. IV): We generalize Eén and Biere’s refinement to first-order logic, resulting in defined predicate elimination, and explain how both types of predicate elimination can be used during saturation.

The last technique we study is blocked clause elimination (Sect. V). It is used in both SAT [14] and QBF solvers [15]. Its generalization to first-order logic has produced good results when used as a preprocessor, especially on satisfiable problems [16]. We explore more ways to use blocked clause elimination on satisfiable problems, including using it to establish equisatisfiability with an empty clause set or as an inprocessing rule. Unfortunately, we find that its use as inprocessing can compromise the refutational completeness of superposition.

All techniques are implemented in the Zipperposition prover (Sect. VI), allowing us to ascertain their usefulness (Sect. VII). The best configuration solves 160 additional problems on benchmarks consisting of all 13 495 first-order TPTP theorems [17]. The raw experimental data are publicly available. More details, including all the proofs, can be found in a technical report [18].

II. PRELIMINARIES

A. Clausal First-Order Logic

Our setting is many-sorted, or many-typed, first-order logic [19] with interpreted equality and a distinguished type o. Each variable x is assigned a non-Boolean type, and each symbol f is assigned a tuple (τ₁,...,τₙ,τ) where n ≥ 0, τᵢ are non-Boolean types, and τ is the result type. We distinguish between predicate symbols, with o as the result type, and function symbols. Nullary function symbols are called constants. Terms are either variables x or well-typed applications f(t₁,...,tₙ), or f if n = 0. A term is ground if it contains no variables. We write d₀ or d to abbreviate (a₁,...,aₙ), and fⁿ(s) to abbreviate i-fold application of unary symbol f: fⁿ(x) abbreviates f(f(f(⋯f(x)))).

An atom is an unoriented equation. A literal is an equation s ≈ t or a disequation s ≱ t. For every predicate symbol p, p(⃗s) abbreviates p(⃗s) ≈ ⊤, and ¬p(⃗s) abbreviates p(⃗s) ≱ ⊤, where ⊤ is a distinguished constant of type o. We distinguish between predicate literals (¬)p(⃗s) and functional literals s ≈ t for which s and t are not of type o. Given a literal L, we write ¬L to denote its complement. A clause C is a multiset of literals, written as L₁ ∨ ⋯ ∨ Lₙ and interpreted disjunctively. Clauses are often defined as sets of literals, but superposition needs multisets. Given a clause set N, N↓₂ denotes the subset of its binary clauses: N↓₂ = {L₁ ∨ L₂ | L₁ ∨ L₂ ∈ N}.

¹https://doi.org/10.5281/zenodo.4552499
B. Superposition Provers

Superposition [2] is a calculus for clausal first-order logic that extends ordered resolution [20] with equality reasoning. It is refutationally complete: Given a finite, unsatisfiable clause set, it will eventually derive the empty clause. It is parameterized by a selection function that influences which of a clause’s literals are eligible as the target of inferences. Moreover, it is compatible with the standard redundancy criterion, which can be used to delete a clause C while preserving completeness of the calculus.

The redundancy criterion relies on a well-founded order that compares terms, literals, or clauses. If N is ground, we can delete C if it is entailed by \(-\)-smaller clauses in N. In general, we can delete C if all of its ground instances are entailed by \(-\)-smaller ground instances of other clauses in N. The criterion can be used to delete a clause that is subsumed by another clause (e.g., p(a) ∨ q by p(x)) or to simplify a clause C into C’, which amounts to adding C’ and then deleting C by appealing to redundancy with respect to N \union \{C’\}. Subsumption and simplification are the main inprocessing mechanisms available to superposition provers. Some systems also implement clause splitting [21]–[23].

Superposition provers saturate the input problem with respect to the calculus’s inference rules using the given clause procedure [24], [25]. It partitions the proof state into a passive set \(\mathcal{P}\) and an active set \(\mathcal{A}\). All clauses start in \(\mathcal{P}\). At each iteration of the procedure’s main loop, the prover chooses a clause C from \(\mathcal{P}\), simplifies it, and moves it to \(\mathcal{A}\). Then all inferences between C and active clauses are performed. The resulting clauses are again simplified and put in \(\mathcal{P}\).

III. HIDDEN-LITERAL-BASED ELIMINATION

In propositional logic, binary clauses from a clause set N can be used to efficiently discover literals L, L’ for which the implication L’ \rightarrow\ L is entailed by N’s binary clauses—i.e., N \subseteq L’ \rightarrow\ L. Heule et al. [4] introduced the concept of hidden literals to capture such implications.

Definition 1: Given a propositional literal L and a propositional clause set N, the set of propositional hidden literals for L and N is HL_p(L, N) = \{L’ \mid L’ \leftrightarrow_p L\} \setminus \{L\}, where \(\leftrightarrow_p\) is defined such that \(\neg L’ \leftrightarrow_p L\) whenever L’ \vee L \in N. Moreover, HL_p(L_1 \vee \cdots \vee L_n, N) = \bigsqcup_{i=1}^n HL_p(L_i, N).

Heule et al. used a fixpoint computation, but our definition based on the reflexive transitive closure is equivalent. Intuitively, a hidden literal can be added to or removed from a clause without affecting its semantics in models of N. By eliminating hidden literals from C, we simplify it. By adding hidden literals to C, we might get a tautology C’ (i.e., a valid clause: \(\models C’\)), meaning that N \subseteq C, thereby enabling us to delete C. Note that HL_p(L, N) is finite for a finite N.

Definition 2: Given L’ \lor L \lor C \in N, if L’ \in HL_p(L, N), hidden literal elimination (HLE) replaces N by (N \setminus \{L’ \lor L \lor C\}) \cup \{L \lor C\}. Given C \in N, \{L_1, \ldots, L_n\} = HL_p(C, N), and C’ = C \lor L_1 \lor \cdots \lor L_n, if C’ is a tautology, hidden tautology elimination (HTE) replaces N by N \setminus \{C\}.

We generalize hidden literals to first-order logic with equality by considering substitutivity of variables as well as congruence of equality.

Definition 3: Given a literal L and a clause set N, the set of hidden literals for L and N is HL(L, N) = \{L’ \mid L’ \leftrightarrow NL\} \setminus \{L\}, where \(\leftrightarrow_{NL}\) is defined so that (1) \(\neg L’ \leftrightarrow_{NL} L\) if L’ \lor L \in N and \(\sigma\) is a substitution; (2) \(s = t \leftrightarrow u[s] = u[t]\) for all terms s, t and contexts u[ ]; and (3) \(u[s] \neq u[t] \leftrightarrow s \neq t\) for all terms s, t and contexts u[ ]. Moreover, HL(L_1 \lor \cdots \lor L_n, N) = \bigsqcup_{i=1}^n HL(L_i, N).

The generalized definition also enjoys the key property that L’ \in HL(L, N) implies N \subseteq L’ \rightarrow\ L. However, HL(L, N) may be infinite even for predicate literals; for example, p(f(x)) \in HL(p(x), \{p(x) \lor \neg p(f(x))\}) for every i.

Based on Definition 3, we can generalize hidden literal elimination and support a related technique:

\[
\begin{align*}
\frac{L’ \lor L \lor C}{L \lor C} & \quad \text{HLE if } L’ \in HL(L, N) \\
\frac{L \lor C}{C} & \quad \text{FLE if } L’, \neg L’ \in HL(\neg L, N)
\end{align*}
\]

Double lines denote simplification rules: When the premises appear in the clause set, the prover can use the redundancy criterion to replace them by the conclusions. The second rule is called failed literal elimination, inspired by the SAT technique of asserting \(\neg L\) if L is a failed literal [5]. Both rules are sound.

Example 4: Consider the clause set N = \{p(x) \lor \neg p(f(x)), p(f(f(x))) \lor a \approx b\} and the clause C = f(a) \neq f(b) \lor p(x). The first clause in N induces p(f(x)) \leftrightarrow p(x), p(f(f(x))) \leftrightarrow p(f(x)), and hence p(f(f(x))) \leftrightarrow p(x). Together with the second clause in N, it can be used to derive a \approx b \rightarrow\ p(x). Finally, using rule (3) of Definition 3, we derive f(a) \neq f(b) \rightarrow\ p(x)—that is, f(a) \neq f(b) \in HL(p(x), N). This allows us to remove C’s first literal using HLE.

Hidden literals can be combined with unit clauses L’ to remove more literals:

\[
\frac{L’ \lor L \lor C}{L’} \quad \text{UnITHLE if } L’ \sigma \in HL(\neg L, N)
\]

Given a unit clause L’ \in N, the rule uses it to discharge L’ \sigma in N \models L’ \rightarrow\ L. As a result, we have N \models \neg L, making it possible to remove L from L’ \lor C.

Example 5: Consider the clause set N = \{p(x) \lor q(f(x)), \neg q(f(a)) \lor f(b) \approx g(c), f(x) \approx g(y)\} and the clause C = \neg p(a) \lor \neg q(b). The first clause in N induces \neg q(f(a)) \leftrightarrow p(a), whereas the second one induces f(b) \approx g(c) \leftrightarrow \neg q(f(a)). Thus, we have f(b) \neq g(c) \leftrightarrow p(a)—that is, f(b) \neq f(c) \in HL(p(x), N). By applying the substitution \(x \rightarrow b, y \rightarrow c\) to the third clause in N, we can fulfill the conditions of UnITHLE and remove C’s first literal.

Next, we generalize hidden tautologies to first-order logic.
Definition 6: A clause \( C \) is a hidden tautology for a clause set \( N \) if there exists a finite set \( \{L_1, \ldots, L_n\} \subseteq \text{HL}(C, N) \) such that \( C \lor L_1 \lor \cdots \lor L_n \) is a tautology.

Example 7: In general, hidden tautologies are not redundant and cannot be deleted during saturation. Consider the unsatisfiable set \( N = \{ \neg a, \neg b, a \lor c, b \lor \neg c \} \), the order \( a < b < c \), and the empty selection function. The only possible superposition inference from \( N \) is between the last two clauses, yielding the hidden tautology \( a \lor b \) (after simplifying away \( T \neq \top \)), which is entailed by the larger clauses \( a \lor c \) and \( b \lor \neg c \). If this clause is removed, the prover could enter an infinite loop, forever generating and deleting the hidden tautology.

To delete hidden tautologies during saturation, the prover could check that all the relevant clause instances encountered along the computation of \( \text{HL} \) are \( \ll \)-smaller than a given hidden tautology. However, this would be expensive and seldom succeed, given that superposition creates lots of nonredundant hidden tautologies. Instead, we propose to simplify hidden tautologies using the following rules:

\[
\begin{align*}
    L \lor L' \lor C & \quad \text{HTR if } \lnot L' \in \text{HL}(L, N) \\
    L \lor C & \quad \text{FLR if } L', \lnot L' \in \text{HL}(L, N)
\end{align*}
\]

We call these techniques hidden tautology reduction and failed literal reduction. Both rules are sound. As with hidden literals, unit clauses \( L' \) can be exploited:

\[
\begin{align*}
    L' \quad L \lor C & \quad \text{UNIT HTR if } L' \sigma \in \text{HL}(L, N) \\
    L' \quad L & \quad \text{UNIT HTR if } L' \sigma \in \text{HL}(L, N)
\end{align*}
\]

We give the simplification rules above the collective name of hidden-literal-based elimination (HLBE). Yet another use of hidden literals is for equivalent literal substitution [3]: If both \( L' \in \text{HL}(L, N) \) and \( L \in \text{HL}(L', N) \), we can often simplify \( L' \sigma \) to \( L \sigma \) in \( N \) if \( L' \sigma \gg L \sigma \). We want to investigate this further.

IV. Predicate Elimination

For propositional logic, variable elimination [10] is one of the main preprocessing and inprocessing techniques. Following the ideas of Gabbay and Olivetti [11], Khasiamshvili and Korovin [13] generalized variable elimination to first-order logic with equality and demonstrated that it is effective as a preprocessor. We propose an improvement that makes this applicable in more cases and show that, with a minor restriction, it can be integrated in a superposition prover without compromising its refutational completeness.

Definition 8: A predicate symbol is called singular (or “nonself-referential”) for a clause set \( N \) if it occurs at most once in every clause contained in \( N \).

Definition 9: Let \( C = p(t_n) \lor C' \) and \( D = \lnot p(t_n) \lor D' \) be clauses with no variables in common. The clause \( s_1 \neq t_1 \lor \cdots \lor s_n \neq t_n \lor C' \lor D' \) is a flat resolvent of \( C \) and \( D \) on \( p \).

Given two clause sets \( M, N \), predicate elimination iteratively replaces clauses from \( N \) containing the symbol \( p \) with all flat resolvents against clauses in \( M \). Eventually, it yields a set with no occurrences of \( p \).

Definition 10: Let \( M, N \) be clause sets and \( p \) be a singular predicate for \( M \). Let \( \rightsquigarrow \) be the following relation on clause set pairs and clause sets:

1. \( \langle M, \{ \lnot p(\bar{s}) \lor C' \} \cup N \rangle \rightsquigarrow \langle M, N' \cup N \rangle \) if \( N' \) is the set that consists of all clauses (up to variable renaming) that are flat resolvents with \( \lnot p(\bar{s}) \lor C' \) on \( p \) and a clause from \( M \) as premises. The premises’ variables are renamed apart.

2. \( \langle M, N \rangle \rightsquigarrow N \) if \( N \) has no occurrences of \( p \).

The resolved set \( M \not\exists p N \) is the clause set \( N' \) such that \( \langle M, N \rangle \rightsquigarrow N' \).

The relation \( \rightsquigarrow \) is confluent up to variable renaming. Thanks to the singularity constraint on \( M \), it also terminates on finite sets because the following ordinal measure decreases:

\[
\nu(\{D_1, \ldots, D_n\}) = \omega^{|D_1|} \oplus \cdots \oplus \omega^{|D_n|}
\]

where \( \nu(D) \) counts the occurrences of \( p \) in \( D \) and \( \oplus \) is the Hessenberg, or natural, sum, which is commutative. For every transition \( \langle M, \{C\} \rangle \rightsquigarrow \langle M, N' \cup N \rangle \), we have \( \nu(\{C\}) = \omega^{|C|} > \omega^{|C| - 1} \cdot |N'| = \nu(N') \).

Definition 11: Let \( N \) be a clause set and \( p \) be a singular predicate for \( N \). Let \( N^p_+ \) consist of all clauses of the form \( \lnot p(\bar{s}) \lor C' \in N \), let \( N^p_- \) consist of all clauses of the form \( \lnot p(\bar{s}) \lor C' \in N \), let \( N_p = N^p_+ \cup N^p_- \), and let \( N_p = N \setminus N_p \).

Definition 12: Let \( N \) be a clause set and \( p \) be a singular predicate for \( N \). Singular predicate elimination (SPE) of \( p \) in \( N \) replaces \( N \) by \( N_p \cup (N^+_p \setminus N^p_-) \).

The result of SPE is satisfiable if and only if \( N \) is satisfiable [13, Theorem 1], justifying SPE’s use in a preprocessor. However, eliminating singular predicates aggressively can dramatically increase the number of clauses. To prevent this, Khasiamshvili and Korovin suggested to replace \( N \) by \( N' \) only if \( \lambda(N') \leq \lambda(N) \) and \( \mu(N') \leq \mu(N) \), where \( \lambda(N) \) is the number of literals in \( N \) and \( \mu(N) \) is the sum for all clauses \( C \in N \) of the square of the number of distinct variables in \( C \).

Compared with what modern SAT solvers use, this criterion is fairly restrictive. We relax it to make it possible to eliminate more predicates, within reason. Let \( K_{\text{tol}} \in \mathbb{N} \) be a tolerance parameter. A predicate elimination step from \( N \) to \( N' \) is allowed if \( \lambda(N') < \lambda(N) + K_{\text{tol}} \) or \( \mu(N') < \mu(N) \) or \( |N'| < |N| + K_{\text{tol}} \).

SPE is effective, but an important refinement has not yet been adapted to first-order logic: variable elimination by substitution. Eén and Biere [10] discovered that a propositional variable \( x \) can be eliminated without computing all resolvents if it is expressible as an equivalence \( x \leftrightarrow \varphi \), where \( \varphi \), the “gate,” is an arbitrary formula that does not reference \( x \). They partition a set \( N \) into a definition set \( G \), essentially the clausification of \( x \leftrightarrow \varphi \), and \( R = N_p \setminus G \), the remaining clauses containing \( p \). To eliminate \( x \) from \( N \) while preserving satisfiability, it suffices to resolve clauses from \( G \) against clauses from \( R \), effectively substituting \( \varphi \) for \( x \) in \( R \). Crucially,
we do not need to resolve pairs of clauses from \( G \) or pairs of
clauses from \( R \). We generalize this idea to first-order logic.

**Definition 13:** Let \( G \) be a clause set, \( p \) be a predicate symbol,
and \( \vec{x} \) be distinct variables. The set \( G \) is a **definition set** for \( p \)
if (1) \( p \) is singular for \( G \), (2) \( G \) consists of clauses of the form
\((-p(\vec{x}) \lor C') \) (up to variable renaming), (3) the variables in \( C' \)
are all among \( \vec{x} \), (4) all clauses in \( G^+ \times p \times G^- \) are tautologies,
and (5) \( E(\vec{x}) \) is unsatisfiable, where the environment \( E(\vec{x}) \)
consists of all subclasses \( C' \) of any \((-p(\vec{x}) \lor C') \in G \) and \( \vec{x} \)
is a tuple of distinct fresh constants substituted in for \( \vec{x} \).

A definition set \( G \) corresponds intuitively to a definition by
cases—e.g.,

\[
p(\vec{x}) = \begin{cases} 
\top & \text{if } \phi(\vec{x}) \\
\bot & \text{if } \psi(\vec{x}) 
\end{cases}
\]

Part (4) states that the case conditions are mutually exclusive
(e.g., \(-\phi(\vec{x}) \lor -\psi(\vec{x}) \)), and part (5) states that they are exhaustive
(e.g., \( \vec{x} \in \neg \phi(\vec{x}) \lor -\psi(\vec{x}) \)). Given a quantifier-free formula
\( p(\vec{x}) \leftrightarrow \phi(\vec{x}) \) with distinct variables \( \vec{x} \) such that \( \phi(\vec{x}) \) does not contain \( p \), any reasonable clasification algorithm would
produce a definition set for \( p \).

**Example 14:** Given the formula \( p(x) \leftrightarrow (q(x) \land (r(x) \lor s(x))) \),
a standard clausification algorithm [26] produces \{\(-p(x) \lor q(x), \neg p(x) \lor r(x) \lor s(x), p(x) \lor \neg q(x) \lor \neg r(x), p(x) \lor
\neg q(x) \lor \neg s(x)\} \), which qualifies as a definition set for \( p \).

Definition sets gracefully generalize gates. They can be recognized
syntactically for formulas such as \( p(\vec{x}) \leftrightarrow \bigvee \{q_i(\vec{s}_i) \}
\) or \( p(\vec{x}) \leftrightarrow \bigwedge \{q_i(\vec{s}_i) \} \), or semantically: Condition (4) can be
checked using the congruence closure algorithm, and condition
(5) amounts to a propositional unsatisfiability check.

The key result about gates carries over to definition sets.

**Definition 15:** Let \( N \) be a clause set, \( p \) be a predicate symbol,
\( G \subseteq N \) be a definition set for \( p \), and \( R = N_p \setminus G \).
**Defined predicate elimination** (DPE) of \( p \) in \( N \) replaces \( N \) by
\( N_p \cup \{p(\vec{s}) \mid \phi(\vec{s}) \} \).

**Theorem 16:** The result of applying DPE to a clause set \( N \)
is satisfiable if and only if \( N \) is satisfiable.

Since there will typically be at most only a few defined predicates in the problem, it makes sense to fall back on SPE
when no definition is found.

**Definition 17:** Let \( N \) be a clause set and \( p \) be a predicate symbol.
If there exists a definition set \( \bar{G} \subseteq N \) for \( p \), **portfolio predicate elimination**
(PPE) on \( p \) in \( N \) replaces \( N \) with \( N_p \cup \{p(\vec{s}) \mid \phi(\vec{s}) \} \), where \( R = N_p \setminus G \). Otherwise, if \( p \) is singular
in \( N \), it results in \( N_p \cup \{p^+ \times p \times N_p^- \} \). In all other cases, it is
not applicable.

Hidden-literal-based techniques fit within the traditional
framework of saturation, because they delete or reduce a clause
based on the **presence** of other clauses. In contrast, predicate elimination relies on the **absence** of clauses from the proof
state. We can still integrate it with superposition as follows:
At every \( k \)th iteration of the given clause procedure, perform
predicate elimination on \( \mathcal{N} \cup \mathcal{P} \), and add all new clauses to \( \mathcal{P} \).

One may wonder whether such an approach preserves the
refutational completeness of the calculus. The answer is no.

To see why, consider a **binary splitting** (BS) rule [21] that
replaces the premise \( C \lor D \) with the conclusions \( p \lor C \) and
\(-p \lor D \), where \( p \) and \( D \) share no free variables, \( p \) is
fresh, and \( p \propto C, D \). This simplification rule can be repeatedly
undone by predicate elimination, leading to troublesome loops:
BS, SPE, BS, SPE,... This breaks completeness.

Our solution is to curtail the entailment relation used by the
redundancy criterion to disallow splitting-like simplifications.
Weak entailment \( \models^b \) is defined via an ad hoc nonclassical
logic so that \( \{p \lor C, \neg p \lor C\} \models^b \{C\} \) and yet \( \models \{p \lor \neg p\} \).
More precisely, this logic is defined via an encoding: \( M \models^b N \)
if and only if \( M^b \models N^b \), where \( p(i)^b = p(i) \neq \bot \), \(-p(i)^b \)
and \( L^b = L \) otherwise. Moreover, the type \( o \) may be interpreted as any set of
arity of at least 2, and \( \bot \) must be a distinguished symbol interpreted differently from \( \top \).

The standard redundancy criterion **Red** based on \( \models^b \)
supports all the familiar deletion and simplification techniques
except splitting. Using **Red** not only prevents looping, but it
also enables the use of the given clause procedure, because
any redundant inference according to **Red** remains redundant
after SPE or DPE. As usual, the devil is in the details, and the
details are in the report [18].

V. SATISFIABILITY BY CLAUSE ELIMINATION

The main approaches to show satisfiability of a first-order
problem are to produce either a finite Herbrand model or
a saturated clause set. Saturations rarely occur except for
very small problems or within decidable fragments. In this
section, we explore an alternative approach that establishes
satisfiability by iteratively removing clauses while preserving
unsatisfiability, until the clause set has been transformed
into the empty set. So far, this technique has been studied
only for QBF [27]. We show that **blocked clause elimination**
(BCE) can be used for this purpose. It can efficiently solve
some problems for which the saturated set would be infinite.
However, it can break the refutational completeness of a
satisfaction proof. We conclude with a procedure that
transforms a finite Herbrand model into a sequence of clause
elimination steps ending in the empty clause set, thereby
demonstrating the theoretical power of clause elimination.

Kiesl et al. [16] generalized blocked clause elimination to
first-order logic. Their generalization uses flat \( L \)-resolvents,
an extension of flat resolvents that resolves a single literal \( L \)
against \( m \) literals of the other clause.

**Definition 18:** Let \( C = L \lor C' \) and \( D = L_1 \lor \cdots \lor L_m \lor D' \),
where (1) \( m \geq 1 \), (2) the literals \( L_i \) are of opposite polarity to
\( L \), (3) \( L \)'s atom is \( p(\vec{x}) \), (4) \( L_i \)'s atom is \( p(i) \) for each \( i \),
and (5) \( C \) and \( D \) have no variables in common. The clause
\( (\bigvee_{i=1}^{m} \bigvee_{j=1}^{m} \bigwedge_{i \neq j}^m) \lor C \lor D' \) is a **flat \( L \)-resolvent** of \( C \) and \( D \).

**Definition 19:** A clause \( C = L \lor C' \) is **(equality/-blocked)**
by \( L \) in a clause set \( N \) if all flat \( L \)-resolvents between \( C \) and
clauses in \( N \setminus \{C\} \) are tautologies.

Removing a blocked clause from a set preserves unsatis-
fiability [16]. Kiesl et al. evaluated the effect of removing
all blocked clauses as a preprocessing step and found that it increases prover’s success rate.

In fact, there exist satisfiable problems that cannot be saturated in finitely many steps regardless of the calculus’s parameters but that can be reduced to an empty, vacuously satisfiable problem through blocked clause elimination.

Example 20: Consider the clause set $N$ consisting of $C = \{p(x,x)\}$ and $D = \neg p(y_1,y_3) \lor p(y_1,y_2) \lor p(y_2,y_3)$. Eventually, the superposition of $p(x,x)$ into $D$’s negative literal needs to be performed, regardless of chosen selection function or term order, with the condition $E_1 = p(z_1,z_2) \lor p(z_2,z_3)$. Then, superposition of $E_1$ into $D$ yields $E_2 = p(z_1,z_2) \lor p(z_2,z_3) \lor p(z_3,z_1)$. Repeating this process yields infinitely many clauses $E_i = p(z_1,z_2) \lor \cdots \lor p(z_i,z_{i+1}) \lor p(z_{i+1},z_1)$ that cannot be eliminated using standard redundancy-based techniques.

In the example above, the clause $D$ is blocked by its second or third literal. If we delete $D$, $C$ becomes blocked in turn. Deleting $C$ leaves us with the empty set, which is vacuously satisfiable. The example suggests that using BCE during saturation might help focus the proof search. Indeed, Kiesl et al. ended their investigations by asking whether BCE can be used as an inprocessing technique in a saturation prover. Unfortunately, in general the answer is no.

Example 21: Consider the unsatisfiable set $N = \{C_1, \ldots, C_6\}$, where

$$
C_1 = \neg c \lor e \lor \neg a \\
C_2 = \neg c \lor \neg e \\
C_3 = b \lor c \\
C_4 = \neg b \lor \neg c \\
C_5 = a \lor b \\
C_6 = c \lor \neg b
$$

Assume the simplification ordering $a < b < c < d < e$ and the selection function that chooses the last negative literal of a clause as presented. Gray boxes indicate literals that can take part in superposition inferences. Only two superposition inferences are possible: from $C_3$ into $C_4$, yielding the tautology $C_7 = b \lor \neg b$, and from $C_5$ into $C_6$, yielding $C_8 = a \lor c$. Clause $C_1$ is clearly redundant, whereas $C_5$ is blocked by its first literal. If we allow removing blocked clauses, the prover enters a loop: $C_5$ is repeatedly generated and deleted.

Although using BCE as inprocessing breaks the completeness of superposition in general, it is conceivable that a well-behaved fragment of BCE might exist. This could be investigated further.

Not only can BCE prevent infinite saturation (Example 20), but it can also be used to convert a finite Herbrand model into a certificate of clause set satisfiability. The certificate uses only blocked clause elimination and addition, in conjunction with a transformation to reduce the clause set to an empty set. This theoretical result explores the relationship between Herbrand models and satisfiability certificates based on clause elimination and addition. It is conceivable that it can form the basis of an efficient way to certify Herbrand models.

In propositional logic, asymmetric literals can be added to or removed from clauses, retaining the equivalence of the resulting clause set with the original one. Kiesl and Suda [28] described an extension of this technique to first-order logic. Their definition of asymmetric literals can be relaxed to allow the addition of more literals, but the resulting set is then only equisatisfiable to the original one, not equivalent. This in turn allows us to show that a problem is satisfiable by reducing it to an empty problem, as is done in some SAT solvers.

For the rest of this section, we work with clausal first-order logic without equality. We use Herbrand models as canonical representatives of first-order models, recalling that every satisfiable set has a Herbrand model [29, Sect. 5.4].

Definition 22: A literal $L$ is a global asymmetric literal (GAL) for a clause $C$ and a clause set $N$ if for every ground instance $C_r$ of $C$, there exists a ground instance $D_0 \lor L'$ of $D \lor L' \in N \setminus \{C\}$ such that $D_0 \subseteq C_r$ and $\neg L' = L_r$.

Theorem 23: If $L$ is a GAL for the clause $C$ and the clause set $N$, then the set $(N \setminus \{C\}) \cup \{C \lor L\}$ is satisfiable if and only if $N$ is satisfiable.

For first-order logic without equality, a clause $C \lor L$ is blocked if all its $L$-resolvents are tautologies [16]. The $L$-resolvent between $C \lor L$ and $\neg L_1 \lor \cdots \lor \neg L_n \lor D$ is $(C \lor D)\sigma$, where $\sigma$ is the most general unifier of the literals $L, L_1, \ldots, L_n$ [20]. Given a Herbrand model $J$ of a problem, the following procedure removes all clauses while preserving satisfiability:

1) Let $q$ be a fresh predicate symbol. For each atom $p(x)$ in the Herbrand universe: If $J \models p(x)$, add the clause $q \lor p(x)$; otherwise, add $q \lor \neg p(x)$. Adding either clause preserves satisfiability as both are blocked by $q$.

2) Since $J$ is a model, for each ground instance $C_r$, there exists a clause $q \lor L$ with $L \in C_r$. We can transform $C \in N$ into $C \lor \neg q$, since $\neg q$ is a GAL for $C$ and $N$.

3) Consider the clause $q \lor L$ added by step 1. Since $L$ is ground and no clause $q \lor \neg L$ was added (since $J$ is a model), the only $L$-resolvents are against clauses added by step 2. Since all of those clauses contain $\neg q$, the resolvents are tautologies. Thus, each $q \lor L$ is blocked and can be removed in turn.

4) The remaining clauses all contain $\neg q$. They can be removed by BCE as well.

The procedure is limited to the first-order logic without equality, since step 3 is justified only if $L$ is a predicate literal. (Otherwise, $L$ cannot block clause $q \lor L$ [16].) The procedure also terminates only for finite Herbrand models.

Example 24: Consider the satisfiable clause set $N = \{r(x) \lor s(x), \neg r(a), \neg s(b)\}$ and a Herbrand model $J$ over $\{a, b, r, s\}$ such that $r(b)$ and $s(a)$ are the only true atoms in $J$. We show how to remove all clauses in $N$ using $J$ by following the procedure above.

Let $N_3 = \{q \lor \neg r(a), q \lor r(b), q \lor s(a), q \lor \neg s(b)\}$. We set $N \leftarrow N \cup N_3$. This preserves satisfiability since all clauses in $N_3$ are blocked. It is easy to check that $\neg q$ is GAL for every clause in $N \setminus N_3$. The only substitutions that need to be considered are $\{x \mapsto a\}$ and $\{x \mapsto b\}$ for $r(x) \lor s(x)$. So we set $N \leftarrow \{\neg q \lor r(x) \lor s(x), \neg q \lor \neg r(a), \neg q \lor \neg s(b)\} \cup N_3$. Clearly, all clauses in $N_3$ are blocked, so we set $N \leftarrow N \setminus N_3$. All clauses remaining in $N$ have a literal $\neg q$ and can be removed, leaving $N$ empty as desired.
VI. IMPLEMENTATION

Hidden-literal-based, predicate, and blocked clause elimination all admit efficient implementations in a superposition prover. In this section, we describe how to implement the first two sets of techniques. For BCE, we refer to Kisel et al. [16]. All techniques are implemented in Zipperposition [30].

A. Hidden-Literal-Based Elimination

For HLBE, an efficient representation of HL($L, N$) is crucial. Because this set may be infinite, we underapproximate it by restricting the length of the transitive chains via a parameter $K_{len}$. Given the current clause set $N$, the finite map $Imp[L']$ associates with each literal $L'$ a set of pairs $(L, M)$ such that $L' \leadsto^k L$, where $k \leq K_{len}$ and $M$ is the multiset of clauses used to derive $L' \leadsto^k L$. Moreover, we consider only transitions of type (1) (as per Definition 3). The following algorithm and deletes clauses. It depends on the global variable $Imp$ and the parameters $K_{len}$ and $K_{imp}$.

\begin{algorithm}
\begin{algorithmic}
\Procedure{AddImplication}{$L_a, L_c, C$}
\If{$Imp[L_a \sigma] \neq \emptyset$ for some renaming $\sigma$}
\State $(L_a, L_c) \leftarrow (L_a \sigma, L_c)$
\EndIf
\If{there are no $L, L', M, \sigma$ such that $(L', M) \in Imp[L]$, $L\sigma = L_c$, and $L' \sigma = L_a$}
\ForAll{$(\sigma, M)$ such that $(L_\sigma \sigma, M) \in Imp[L_a] \sigma$}
\State erase all $(L', M')$ such that $M \subseteq M'$ from $Imp[L_a \sigma]$
\EndFor
\EndIf
\ForAll{$L$ such that $(L', M) \in Imp[L]$
\State $L_a \sigma = L'$ for some $\sigma$
\If{$|M| < K_{len}$}
\State $Imp[L] \leftarrow Imp[L] \cup \{(L_a \sigma, M \uplus \{C\})\}$
\EndIf
\EndFor
\State $(L', M) \in Imp[L], |M| < K_{len}$
\State $Imp[L_a \sigma] \leftarrow Imp[L_{\sigma}] \cup Concl$
\State $Concl \leftarrow \{(s \neq t, \{C\}) \mid \exists u. L_c = u[s] \neq u[t]\}$
\State $Imp[L_a \sigma] \leftarrow Imp[L_a \sigma] \cup \{(L_\sigma, \{C\})\} \cup Concl$
\EndProcedure
\Procedure{TrackClause}{$C$}
\If{$C = L_1 \lor L_2$}
\State AddImplication($\neg L_1, L_2, C$)
\State AddImplication($\neg L_2, L_1, C$)
\EndIf
\If{$L_2 = \neg L_1 \sigma$ for some nonidempotent $\sigma$}
\ForAll{$i \leftarrow 1$ to $K_{imp}$}
\State $L_2 \leftarrow L_2 \sigma$
\EndFor
\EndIf
\EndProcedure
\Procedure{UntrackClause}{$C$}
\ForAll{$L_a, L_c, M$ such that $(L_c, M) \in Imp[L_a]$}
\If{$C \in M$}
\State erase $(L_c, M)$ from $Imp[L_a]$
\EndIf
\EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm}

The algorithm views a clause $L \lor L'$ as two implications $\neg L \rightarrow L'$ and $\neg L' \rightarrow L$. It stores only one entry for all literals equal up to variable renaming (line 2). Each implication $L_a \rightarrow L_c$ represented by the clause is stored only if its generalization is not present in $Imp$ (line 4). Conversely, all instances of the implication are removed (line 6).

Next, the algorithm finds each implication stored in $Imp$ that can be linked to $L_a \rightarrow L_c$: either $L_a$ becomes the new consequent (line 9) or $L_a$ becomes the new antecedent (line 13). If $L_c$ can be decomposed into $u[s] \neq u[t]$, rule (3) of Definition 3 allows us to store $s \neq t$ in $Imp[L_a]$ (line 18). This is an exception to the idea that transitive chains should only use rule (1). The application of rule (3) does not count toward the bound $K_{len}$.

In first-order logic, different instances of the same clause can be used along a transitive chain. For example, the clause $C = \neg p(x) \lor p(f(x))$ induces $p(x) \rightarrow p(f^i(x))$ for all $i$. The algorithm discovers such self-implications (line 23): For each clause $C$ of the form $\neg L \lor L\sigma$, where $\sigma$ is nonidempotent, the entrees $(L, \{C\}), (L\sigma, \{C\})$ are added to $Imp[L]$, where $K_{imp}$ is a parameter.

To track and untrack clauses efficiently, we implement the mapping $Imp$ as a nonperfect discrimination tree [31]. Given a query literal $L$, this indexing data structure efficiently finds all literals $L'$ such that for some $\sigma$, $L'\sigma = L$ and $Imp[L'] \neq \emptyset$. We can use it to optimize all lookups except the one on line 9. For this remaining lookup, we add an index $Imp^{-1}$ that inverts $Imp$, i.e., $Imp^{-1}[L] = \{L' \mid Imp[L'] = (L, M)$ for some $M\}$. To avoid sequentially going through all entries in $Imp$ when the prover deletes them, for each clause $C$ we keep track of each literal $L$ such that $C$ appears in $Imp[L]$.

Rules HLE and HTR have a simple implementation based on lookups in $Imp$. To implement UNITHLE and UNITHTR we maintain the index $Unit$, containing literals $L_\sigma \sigma$, such that $(L_\sigma, M) \in Imp[L_a]$ for some $M$ and $L_a$ and $\sigma$ is the most general unifier of $L'$ and $L_a$, for some unit clause $\{L'\}$. Implementation of FLE and FLR also uses $Unit$: when $(L', M)$ is added to $Imp[L]$, we check if $(\neg L', M') \in Imp[L]$ for some $M'$. If so, $\neg L$ is added to $Unit$.

In propositional logic, the most efficient known approach constructs the binary implication graph for the clause set $N$ [4], with edges $(\neg L, L')$ and $(\neg L', L)$ whenever $L \lor L' \in N$. To avoid traversing the graph repeatedly, solvers rely on timeouts to discover connections between literals. This relies on syntactic comparisons of literals, which is very efficient in propositional logic but not in first-order logic, because of substitutions and congruence.

B. Predicate Elimination

To implement portfolio predicate elimination, we maintain a record for each predicate symbol $p$ occurring in the problem with the following fields: set of definition clauses for $p$, set of nondefinition clauses in which $p$ occurs once, and set of clauses in which $p$ occurs more than once. These records are kept in a priority queue, prioritized by properties such as presence of definition sets and number of estimated resolutions. If $p$ is the highest-priority symbol that is eligible for SPE or DPE, we eliminate it by removing all the clauses stored in $p$'s record from the proof state and by adding flat resolvents to the passive set. Eliminating a symbol might make another symbol eligible.
As an optimization, predicate elimination keeps track only of symbols that appear at most $K_{occ}$ times in the clause set. For inprocessing, we use signals that Zipperposition emits whenever a clause is added to or removed from the proof state and update the records accordingly. At the beginning of the 1st, $(K_{iter} + 1)_{st}$, $(2K_{iter} + 1)_{st}$, … iteration of the given clause procedure’s loop body, predicate elimination is systematically applied to the entire proof state. The first application of inprocessing amounts to preprocessing. By default, $K_{occ} = 512$ and $K_{iter} = 10$. The same ideas and limits apply for blocked clause elimination.

Zipperposition uses its integrated SAT solver to check the condition (5) of Definition 13. During experimentation, we noticed that recognizing definitions of symbols that occur in the conjecture often harms performance. Thus, Zipperposition recognizes definitions only for nonconjecture symbols.

VII. Evaluation

We measure the impact of our elimination techniques for various values of their parameters. As a baseline, we use Zipperposition’s first-order portfolio mode, which runs the prover in 13 configurations of heuristic parameters in consecutive time slices. None of these configurations use our new techniques. To evaluate a given parameter value, we fix it across all 13 configurations and compare the results with the baseline.

The benchmark set consists of all 13 495 CNF and FOF TPTP 7.3.0 theorems [17]. The experiments were carried out on StarExec servers [32] equipped with Intel Xeon E5-2609 CPUs clocked at 2.40 GHz. The portfolio mode uses a single CPU core with a CPU time limit of 180 s. The base configuration solves 7897 problems. The values in the tables indicate the number of problems solved minus 7897. Thus, positive numbers indicate gains over the baseline. The best result is shown in bold.

A. Hidden-Literal-Based Elimination

The first experiments use all implemented HLBE rules. To avoid overburdening Zipperposition, we can enable an option to limit the number of tracked clauses for hidden literals. Once the limit has been reached, any request for tracking a clause will be rejected until a tracked clause is deleted. We can choose which kind of clauses are tracked: only active, only passive, or both. We also vary the maximal implication chain length $K_{len}$ and the number of computed self-implications $K_{imp}$.

In Zipperposition, every lookup for instances or generalizations of $s \approx t$ must be done once for each orientation of the equation. To avoid this inefficiency, and also because the implementation of hidden literals does not fully exploit congruence, we can disable tracking clauses with at least one functional literal.Clauses containing functional literals can then still be simplified.

Figures 1 and 2 show the results, without and with functional literal tracking enabled, for $K_{len} = 2$ and $K_{imp} = 0$. The results suggest that tracking functional literals is not worth the effort but that tracking predicate literals is. The best improvement is observed when both active and passive clauses are tracked. Normally DISCOUNT-loop provers [25] such as Zipperposition do not simplify active clauses using passive clauses, but here we see that this can be effective. Figure 3 shows the impact of varying $K_{len}$ and $K_{imp}$, when 500 clauses from the entire proof state are tracked. These results suggest that computing long implication chains is counterproductive.

B. Predicate and Blocked Clause Elimination

For defined predicate elimination, the number of resolvents grows exponentially with the number of occurrences of $p$. To avoid this expensive computation, we limit the applicability of PPE to proof states for which $p$ is singular. According to our informal experiments, full PPE, without this restriction, generally performs less well.

Predicate elimination can be done using Khasidashvili and Korovin’s criterion (K&K) or using our relaxed criterion with different values of $K_{tol}$. Figure 4 shows the results for SPE and PPE used as preprocessors. Our numbers corroborate Khasidashvili and Korovin’s findings: SPE with K&K proves 70 more problems than the base, a 0.9% increase, comparable to the 1.8% they observe when they combine SPE with additional preprocessing. Remarkably, the number of additional proved problems more than doubles when we use our criterion with $K_{tol} > 0$, for both SPE and PPE.

Although this is not evident in Figure 4, varying $K_{tol}$ substantially changes the set of problems solved. For example, when $K_{tol} = 0$, SPE proves 60 theorems not proved using $K_{tol} = 50$. The effect weakens as $K_{tol}$ grows. When $K_{tol} = 100$, SPE proves only 13 problems not found when $K_{tol} = 200$. Similarly, the set of problems proved by SPE and PPE differs: When $K_{tol} = 25$, 14 problems are proved by PPE but missed by SPE. Recognizing definition sets is useful: PPE outperforms SPE regardless of the criterion.

Performing BCE and variable elimination until fixpoint increases the performance of SAT solvers [14]. We can check whether the same holds for superposition provers. In this experiment, we use the relaxed criterion with $K_{tol} = 25$ and HLBE which tracks up to 500 clauses from any clause set, $K_{len} = 2$, and $K_{imp} = 0$. We use each technique as preprocessing and inprocessing.
The results are summarized in Figure 5, where the + sign denotes the combination of techniques. We confirm the results obtained by Kiesl et al. about the performance of BCE as preprocessing: It helps prove 30 more problems, increasing the success rate by roughly 0.4%. The same percentage increase was obtained by Kiesl et al. Using BCE as an inprocessing, however, hurts performance, presumably because of its incompatibility with the redundancy criterion.

For preprocessing, the combinations SPE+BCE and PPE+BCE performed roughly on a par with SPE and PPE, respectively. This stands in contrast to the situation with SAT solvers, where such a combination usually helps. It is also worth noting that the inprocessing techniques never outperform their preprocessing counterparts. The last column shows that combining HLBE with other elimination techniques overburdens the prover.

C. Satisfiability by Blocked Clause Elimination

Kiesl et al. found that blocked clause elimination is especially effective on satisfiable problems. To corroborate their results and ascertain whether a combination of predicate elimination and blocked clause elimination increases the success rate, we evaluate BCE on all 2273 satisfiable or TPTP FOF and CNF problems. The hardware and CPU time limits are the same as in the experiments above. Figure 6 presents the results.

The baseline establishes the satisfiability of 856 problems. We consider only preprocessing techniques, since BCE compromises refutational completeness—a saturation does not guarantee that the original problem was satisfiable. We note that recognizing definition sets makes almost no difference on satisfiable problems. The sets of problems solved by BCE and PPE differ—30 problems are solved by BCE and not by PPE.

VIII. Conclusion

We adapted several preprocessing and inprocessing elimination techniques implemented in modern SAT solvers so that they work in a superposition prover. This involved lifting the techniques to first-order logic with equality but also tailoring them to work in tandem with superposition and its redundancy criterion. Although SAT solvers and superposition provers embody radically different philosophies, we found that the lifted SAT techniques provide valuable optimizations.

We see several avenues for future work. First, the implementation of hidden literals could be extended to exploit equality congruence. Second, although inprocessing blocked clause elimination is incomplete in general, we hope to achieve refutational completeness for a substantial fragment of it. Third, predicate and blocked clause elimination, which thrives on the absence of clauses from the proof state, could be enhanced by tagging and ignoring generated clauses that have not yet been used to subsume or simplify untagged clauses. Fourth, predicate and blocked clause elimination could be extended to work with functional literals. Fifth, more SAT techniques could be adapted, including bounded variable addition [33] and blocked clause addition [34]. Sixth, the techniques we covered could be adapted to work with other first-order calculi such as SMT and tableaux, or generalized further to work with higher-order calculi such as combinatorial superposition [35] and λ-superposition [36].

A. Acknowledgment

We are grateful to the maintainers of StarExec for letting us use their service. Uwe Waldmann participated in the search for a counterexample to completeness of BCE as inprocessing and confirmed that Example 21 is correct. He also suggested major simplifications and helped us debug the proofs of the claims about predicate elimination. Anne Baanen helped us define the nonclassical logic used to disallow splitting. Ahmed Bhayat, Armin Biere, Mathias Fleury, Benjamin Kiesl, and the anonymous reviewers made some useful comments on our manuscript, and Mark Summerfield suggested many textual improvements. We thank them all.

Vukmirović and Blanchette’s research has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Blanchette’s research has also received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward). Heule is supported by the National Science Foundation (NSF) under grant CCF-2015445.
REFERENCES


