Formal Methods in Computer-Aided Design 2021

SAT-Inspired Eliminations for Superposition

Petar Vukmirović[1], Jasmin Blanchette[1,2], Marijn J.H. Heule[3]

[1]Vrije Universiteit Amsterdam, Amsterdam, the Netherlands
[2]Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
[3]Carnegie 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, a leading first-order proof calculus, and we strengthen known preprocessing techniques. 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] adapted 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 the proof search as inprocessing.

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.[1] 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 (or sort) o. Each variable x is assigned a non-Boolean type, and each symbol f is assigned a tuple $(\tau_1,\ldots,\tau_n,\tau)$ where $n \geq 0$, τ_i 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 $\mathsf{f}(t_1,\ldots,t_n)$, or f if $n=0$. A term is *ground* if it contains no variables. We assume standard definitions and notations for positions, subterms, and contexts [20]. We abbreviate a vector $(a_1,\ldots,a_n)$ to $\vec{a}_n$ or $\vec{a}$, and write $\mathsf{f}^i(s)$ for the i-fold application of an unary symbol f (e.g., $\mathsf{f}^3(x) = \mathsf{f}(\mathsf{f}(\mathsf{f}(x)))$).

An atom is an equation $s \approx t$ corresponding to an unordered pair $\{s,t\}$. A literal is an equation $s \approx t$ or a disequation $s \not\approx t$. For every predicate symbol p, $\mathsf{p}(\vec{s})$ abbreviates $\mathsf{p}(\vec{s}) \approx \top$, and $\neg\,\mathsf{p}(\vec{s})$ abbreviates $\mathsf{p}(\vec{s}) \not\approx \top$, where $\top$ is a distinguished constant of type o. We distinguish between *predicate literals* $(\neg)\,\mathsf{p}(\vec{s})$ and *functional literals* $s \approx t$, where s and t are not of type o. Given a literal L, we overload notation and write $\neg L$ to denote its complement. A clause C is a multiset of literals,

[1]https://doi.org/10.5281/zenodo.4552499

https://doi.org/

written as $L_1 \vee \cdots \vee L_n$ and interpreted disjunctively. Clauses are often defined as sets of literals, but superposition needs multisets; with multisets, an instance $C\sigma$ always has the same number of literals as C, a most convenient property. Given a clause set N, $N{\downarrow_2}$ denotes the subset of its binary clauses: $N{\downarrow_2} = \{L_1 \vee L_2 \mid L_1 \vee L_2 \in N\}$.

B. Superposition Provers

Superposition [2] is a calculus for clausal first-order logic that extends ordered resolution [21] 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 an order $\succ$ that compares terms, literals, or clauses. The order is used to determine whether clauses can be deleted. If N is ground, C can be deleted if it is entailed by $\prec$-smaller clauses in N. This definition is lifted to nonground sets N. The criterion can be used to delete a clause that is *subsumed* by another clause (e.g., $\mathsf{p}(\mathsf{a}) \vee \mathsf{q}$ by $\mathsf{p}(x)$) or to *simplify* a clause C into C', which amounts to adding C' and then deleting C as redundant with respect to $N \cup \{C'\}$. Subsumption and simplification are the main inprocessing mechanisms available to superposition provers. Some provers also implement clause splitting [22]–[24].

Superposition provers saturate the input problem with respect to the calculus's inference rules using the *given clause procedure* [25], [26]. 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 $L' \longrightarrow L$ is entailed by N's binary clauses—i.e., $N{\downarrow_2} \models L' \longrightarrow L$. 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 $\mathrm{HL}_{\mathrm{p}}(L,N) = \{L' \mid L' \hookrightarrow_{\mathrm{p}}^{*} L\} \setminus \{L\}$, where $\hookrightarrow_{\mathrm{p}}$ is defined such that $\neg L_1 \hookrightarrow_{\mathrm{p}} L_2$ whenever $L_1 \vee L_2 \in N$. Moreover, $\mathrm{HL}_{\mathrm{p}}(L_1 \vee \cdots \vee L_n, N) = \bigcup_{i=1}^{n} \mathrm{HL}_{\mathrm{p}}(L_i, N)$.

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 $N{\downarrow_2} \models C$, thereby enabling us to delete C. Note that $\mathrm{HL}_{\mathrm{p}}(L,N)$ is finite for a finite N.

Definition 2: Given $L' \vee L \vee C \in N$, if $L' \in \mathrm{HL}_{\mathrm{p}}(L,N)$, *hidden literal elimination* (HLE) replaces N by $(N \setminus \{L' \vee L \vee C\}) \cup \{L \vee C\}$. Given $C \in N$, $\{L_1, \ldots, L_n\} = \mathrm{HL}_{\mathrm{p}}(C,N)$, and $C' = C \vee L_1 \vee \cdots \vee L_n$, if C' is a tautology, *hidden tautology elimination* (HTE) replaces N by $N \setminus \{C\}$.

Theorem 3: The result of applying HLE or HTE to a clause set N is equivalent to N.

Proof: For HLE, if $L' \in \mathrm{HL}_{\mathrm{p}}(L,N)$, $N{\downarrow_2} \models \neg L' \vee L$. Then, subsumption resolution yields shortened clause $L \vee C'$ from Definition 2. For HTE, it can be shown that $N' \models C$ if and only if $C \vee L'$, where $L' \in \mathrm{HL}_{\mathrm{p}}(C,N)$. By transitivity of equivalence, we get the desired result. ■

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

Definition 4: Given a literal L and a clause set N, the set of *hidden literals* for L and N is $\mathrm{HL}(L,N) = \{L' \mid L' \hookrightarrow^{*} L\} \setminus \{L\}$, where $\hookrightarrow$ is defined so that (1) $\neg L'\sigma \hookrightarrow L\sigma$ if $L' \vee L \in N$ and σ is a substitution; (2) $s \approx t \hookrightarrow u[s] \approx u[t]$ for all terms s,t and contexts $u[\,]$; and (3) $u[s] \not\approx u[t] \hookrightarrow s \not\approx t$ for all terms s,t and contexts $u[\,]$. Moreover, $\mathrm{HL}(L_1 \vee \cdots \vee L_n, N) = \bigcup_{i=1}^{n} \mathrm{HL}(L_i, N)$.

The generalized definition also enjoys the key property that $L' \in \mathrm{HL}(L,N)$ implies $N{\downarrow_2} \models L' \longrightarrow L$. However, $\mathrm{HL}(L,N)$ may be infinite even for predicate literals; for example, $\mathsf{p}(\mathsf{f}^i(x)) \in \mathrm{HL}(\mathsf{p}(x), \{\mathsf{p}(x) \vee \neg\,\mathsf{p}(\mathsf{f}(x))\})$ for every i.

Based on Definition 4, we can generalize hidden literal elimination and support a related technique:

$$\frac{L' \vee L \vee C}{L \vee C}\,\text{HLE} \quad \text{if } L' \in \mathrm{HL}(L,N)$$

$$\frac{L \vee C}{C}\,\text{FLE} \quad \text{if } L', \neg L' \in \mathrm{HL}(\neg L,N)$$

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]. It is easy to see that rule HLE is sound. From $L' \in \mathrm{HL}(L,N)$ we have $N \models L' \longrightarrow L$ (i.e., $\neg L' \vee L$). Performing subsumption resolution [21] between $L' \vee L \vee C$ and $\neg L' \vee L$ yields the conclusion, which is therefore entailed by N. For FLE, the condition $L', \neg L' \in \mathrm{HL}(\neg L,N)$ means that $N{\downarrow_2} \models \{\neg L' \vee \neg L, L' \vee \neg L\} \models \neg L$.

Example 5: Consider the clause set $N = \{\mathsf{p}(x) \vee \neg\,\mathsf{p}(\mathsf{f}(x)),\ \mathsf{p}(\mathsf{f}(\mathsf{f}(x))) \vee \mathsf{a} \approx \mathsf{b}\}$ and the clause $C = \mathsf{f}(\mathsf{a}) \not\approx \mathsf{f}(\mathsf{b}) \vee \mathsf{p}(x)$. The first clause in N induces $\mathsf{p}(\mathsf{f}(x)) \hookrightarrow \mathsf{p}(x)$, $\mathsf{p}(\mathsf{f}(\mathsf{f}(x))) \hookrightarrow \mathsf{p}(\mathsf{f}(x))$, and hence $\mathsf{p}(\mathsf{f}(\mathsf{f}(x))) \hookrightarrow^{*} \mathsf{p}(x)$. Together with the second clause in N, it can be used to derive $\mathsf{a} \not\approx \mathsf{b} \hookrightarrow^{*} \mathsf{p}(x)$. Finally, using rule (3) of Definition 4, we derive $\mathsf{f}(\mathsf{a}) \not\approx \mathsf{f}(\mathsf{b}) \hookrightarrow^{*} \mathsf{p}(x)$—that is, $\mathsf{f}(\mathsf{a}) \not\approx \mathsf{f}(\mathsf{b}) \in \mathrm{HL}(\mathsf{p}(x), N)$. This allows us to remove C's first literal using HLE.

Two special cases of HLE exploit equality congruence as embodied by conditions (2) and (3) of Definition 4 without requiring to compute the HL set:

$$\frac{s \approx t \vee u[s] \approx u[t] \vee C}{u[s] \approx u[t] \vee C}\,\textsc{CongHLE}^{+}$$

$$\frac{s \not\approx t \vee u[s] \not\approx u[t] \vee C}{s \not\approx t \vee C}\,\textsc{CongHLE}^{-}$$

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

$$\frac{L' \quad L \vee C}{L' \quad C}\,\textsc{UnitHLE} \quad \text{if } L'\sigma \in \mathrm{HL}(\neg L, N)$$

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

Example 6: Consider the clause set $N = \{\mathsf{p}(x) \vee \mathsf{q}(\mathsf{f}(x)),\ \neg\mathsf{q}(\mathsf{f}(\mathsf{a})) \vee \mathsf{f}(\mathsf{b}) \approx \mathsf{g}(\mathsf{c}),\ \mathsf{f}(x) \not\approx \mathsf{g}(y)\}$ and the clause $C = \neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{q}(\mathsf{b})$. The first clause in N induces $\neg\mathsf{q}(\mathsf{f}(\mathsf{a})) \hookrightarrow \mathsf{p}(\mathsf{a})$, whereas the second one induces $\mathsf{f}(\mathsf{b}) \not\approx \mathsf{g}(\mathsf{c}) \hookrightarrow \neg\mathsf{q}(\mathsf{f}(\mathsf{a}))$. Thus, we have $\mathsf{f}(\mathsf{b}) \not\approx \mathsf{g}(\mathsf{c}) \hookrightarrow^{*} \mathsf{p}(\mathsf{a})$—that is, $\mathsf{f}(\mathsf{b}) \not\approx \mathsf{f}(\mathsf{c}) \in \mathrm{HL}(\mathsf{p}(\mathsf{a}), N)$. By applying the substitution $\{x \mapsto \mathsf{b}, y \mapsto \mathsf{c}\}$ 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 7: A clause C is a *hidden tautology* for a clause set N if there exists a finite set $\{L_1, \dots, L_n\} \subseteq \mathrm{HL}(C, N)$ such that $C \vee L_1 \vee \cdots \vee L_n$ is a tautology.

Example 8: In general, hidden tautologies are not redundant and cannot be deleted during saturation. Consider the unsatisfiable set $N = \{\neg\mathsf{a}, \neg\mathsf{b}, \mathsf{a} \vee \mathsf{c}, \mathsf{b} \vee \neg\mathsf{c}\}$, the order $\mathsf{a} \prec \mathsf{b} \prec \mathsf{c}$, and the empty selection function. The only possible superposition inference from N is between the last two clauses, yielding the hidden tautology $\mathsf{a} \vee \mathsf{b}$ (after simplifying away $\top \not\approx \top$), which is entailed by the larger clauses $\mathsf{a} \vee \mathsf{c}$ and $\mathsf{b} \vee \neg\mathsf{c}$. 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 HL are $\prec$-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:

$$\frac{L \vee L' \vee C}{L \vee L'}\,\textsc{HTR} \quad \text{if } \neg L' \in \mathrm{HL}(L, N) \text{ and } C \neq \bot$$

$$\frac{L \vee C}{L}\,\textsc{FLR} \quad \text{if } L', \neg L' \in \mathrm{HL}(L, N) \text{ and } C \neq \bot$$

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

$$\frac{L' \quad L \vee C}{L' \quad L}\,\textsc{UnitHTR} \quad \text{if } L'\sigma \in \mathrm{HL}(L, N) \text{ and } C \neq \bot$$

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 $L' \in \mathrm{HL}(L, N)$ and $L \in \mathrm{HL}(L', N)$, we can often simplify $L'\sigma$ to $L\sigma$ in N if $L'\sigma \succ L\sigma$. We want to investigate this further.

Theorem 9: The rules HLE, FLE, CongHLE^{+}, CongHLE^{-}, UnitHLE, HTR, FLR, and UnitHTR are sound simplification rules.

IV. Predicate Elimination

For propositional logic, variable elimination [10] is one of the main preprocessing and inprocessing techniques. Following Gabbay and Ohlbach's ideas [11], Khasidashvili 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.

A. Singular Predicates

Khasidashvili and Korovin's preprocessing technique removes singular predicates (which they call "non-self-referential predicates") from the problem using so-called flat resolution.

Definition 10: A predicate symbol is called *singular* (or "non-self-referential") for a clause set N if it occurs at most once in every clause contained in N.

Definition 11: Let $C = \mathsf{p}(\vec{s}_n) \vee C'$ and $D = \neg\mathsf{p}(\vec{t}_n) \vee D'$ be clauses with no variables in common. The clause $s_1 \not\approx t_1 \vee \cdots \vee s_n \not\approx t_n \vee C' \vee D'$ is a *flat resolvent* of C and D on p.

Given two (possibly identical) 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 12: 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) $(M, \{(\neg)\,\mathsf{p}(\vec{s}) \vee C'\} \uplus N) \rightsquigarrow (M, N' \cup N)$ if N' is the set that consists of all clauses (up to variable renaming) that are flat resolvents with $(\neg)\,\mathsf{p}(\vec{s}) \vee C'$ on p and a clause from M as premises. The premises' variables are renamed apart.
2) $(M, N) \rightsquigarrow N$ if N has no occurrences of p.

The *resolved set* $M \rtimes_{\mathsf{p}} N$ is the clause set N' such that $(M, N) \rightsquigarrow^{*} N'$.

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: $\nu(\{D_1,\ldots,D_n\}) = \omega^{\nu(D_1)} \oplus \cdots \oplus \omega^{\nu(D_n)}$, where $\nu(D)$ counts the occurrences of p in D, ω is the first infinite ordinal, and $\oplus$ is the Hessenberg, or natural, sum, which is commutative. For every transition $(M, \{C\} \cup N) \rightsquigarrow (M, N' \cup N)$, we have $\nu(\{C\}) = \omega^{\nu(C)} > \omega^{\nu(C)-1} \cdot |N'| = \nu(N')$.

Next, it is useful to partition clause sets into subsets based on the presence and polarity of a singular predicate.

Definition 13: Let N be a clause set and p be a singular predicate for N. Let N^+_p consist of all clauses of the form $\mathsf{p}(\vec{s}) \vee C' \in N$, let N^-_p consist of all clauses of the form $\neg\,\mathsf{p}(\vec{s}) \vee C' \in N$, let $N_\mathsf{p} = N^+_\mathsf{p} \cup N^-_\mathsf{p}$, and let $\overline{N}_\mathsf{p} = N \setminus N_\mathsf{p}$.

Definition 14: 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 $\overline{N}_\mathsf{p} \cup (N^+_\mathsf{p} \rtimes\!\!\!\ltimes_\mathsf{p} N^-_\mathsf{p})$.

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, Khasidashvili and Korovin suggested to replace N by N' only if $\lambda(N') \le \lambda(N)$ and $\mu(N') \le \mu(N)$, 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 $K_\mathrm{tol} \in \mathbb{N}$ be a tolerance parameter. A predicate elimination step from N to N' is allowed if $\lambda(N') < \lambda(N) + K_\mathrm{tol}$ or $\mu(N') < \mu(N)$ or $|N'| < |N| + K_\mathrm{tol}$.

B. Defined Predicates

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 $\mathsf{x} \leftrightarrow \varphi$, where φ, 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 $\mathsf{x} \leftrightarrow \varphi$, and $R = N_\mathsf{p} \setminus G$, 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 φ 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 15: 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 $(\neg)\,\mathsf{p}(\vec{x}) \vee C'$ (up to variable renaming), (3) the variables in C' are all among $\vec{x}$, (4) all clauses in $G^+_\mathsf{p} \rtimes\!\!\!\ltimes_\mathsf{p} G^-_\mathsf{p}$ are tautologies, and (5) $E(\vec{c})$ is unsatisfiable, where the *environment* $E(\vec{x})$ consists of all subclauses C' of any $(\neg)\,\mathsf{p}(\vec{x}) \vee C' \in G$ and $\vec{c}$ is a tuple of distinct fresh constants substituted in for $\vec{x}$.

A definition set G corresponds intuitively to a definition by cases in mathematics—e.g.,

$$\mathsf{p}(\vec{x}) = \begin{cases} \top & \text{if } \varphi(\vec{x}) \\ \bot & \text{if } \psi(\vec{x}) \end{cases}$$

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

Example 16: Given the formula $\mathsf{p}(x) \leftrightarrow \mathsf{q}(x) \wedge (\mathsf{r}(x) \vee \mathsf{s}(x))$, a standard clausification algorithm [27] produces $\{\neg\,\mathsf{p}(x) \vee \mathsf{q}(x),\ \neg\,\mathsf{p}(x) \vee \mathsf{r}(x) \vee \mathsf{s}(x),\ \mathsf{p}(x) \vee \neg\,\mathsf{q}(x) \vee \neg\,\mathsf{r}(x),\ \mathsf{p}(x) \vee \neg\,\mathsf{q}(x) \vee \neg\,\mathsf{s}(x)\}$, which qualifies as a definition set for p.

Definition sets generalize Eén and Biere's gates. They can be recognized syntactically for formulas such as $\mathsf{p}(\vec{x}) \leftrightarrow \bigvee_i \mathsf{q}_i(\vec{s_i})$ or $\mathsf{p}(\vec{x}) \leftrightarrow \bigwedge_i \mathsf{q}_i(\vec{s_i})$, 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 propositional gates carries over to definition sets.

Definition 17: Let N be a clause set, p be a predicate symbol, $G \subseteq N$ be a definition set for p, and $R = N_\mathsf{p} \setminus G$. *Defined predicate elimination* (DPE) of p in N replaces N by $\overline{N}_\mathsf{p} \cup (G_\mathsf{p} \rtimes\!\!\!\ltimes_\mathsf{p} R_\mathsf{p})$.

Theorem 18: 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 19: Let N be a clause set and p be a predicate symbol. If there exists a definition set $G \subseteq N$ for p, *portfolio predicate elimination* (PPE) on p in N replaces N with $\overline{N}_\mathsf{p} \cup (G_\mathsf{p} \rtimes\!\!\!\ltimes_\mathsf{p} R_\mathsf{p})$, where $R = N_\mathsf{p} \setminus G$. Otherwise, if p is singular in N, it results in $\overline{N}_\mathsf{p} \cup (N^+_\mathsf{p} \rtimes\!\!\!\ltimes_\mathsf{p} N^-_\mathsf{p})$. In all other cases, it is not applicable.

C. Refutational Completeness

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 kth iteration of the given clause procedure, perform predicate elimination on $\mathcal{A} \cup \mathcal{P}$, 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 the following *binary splitting* rule based on Riazanov and Voronkov [22]:

$$\frac{C \vee D}{\mathsf{p} \vee C \quad D \vee \neg\,\mathsf{p}}\ \mathrm{BS}$$

Provisos: C and D have no free variables in common, p is fresh, and p is $\prec$-smaller than C and D. Since the conclusions are smaller than the premise, the rule can be applied aggressively as a simplification. But notice that the effect of splitting can be undone by singular predicate elimination, possibly giving rise to 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 $\models^\flat$ is defined via an ad hoc nonclassical logic so that $\{\mathsf{p} \vee C, \neg\,\mathsf{p} \vee C\} \not\models^\flat \{C\}$ and yet $\models^\flat \{\mathsf{p} \vee \neg\,\mathsf{p}\}$. More precisely, this logic is defined via an encoding: $M \models^\flat N$ if and only if $M^\flat \models N^\flat$, where $\mathsf{p}(\vec{t})^\flat = \mathsf{p}(\vec{t}) \not\approx \bot$, $\neg\,\mathsf{p}(\vec{t})^\flat = \mathsf{p}(\vec{t}) \not\approx \top$, and $L^\flat = L$ otherwise. Moreover, the type o may be interpreted as any set of cardinality at least 2, and $\bot$ must be a distinguished symbol interpreted differently from $\top$.

The standard redundancy criterion $\mathit{Red}^\flat$ based on $\models^\flat$ supports all the familiar deletion and simplification techniques except splitting. Using $\mathit{Red}^\flat$ not only prevents looping, but it also enables the use of the given clause procedure, because any redundant inference according to $\mathit{Red}^\flat$ 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 [28]. 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 saturation prover. 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 20: Let $C = L \vee C'$ and $D = L_1 \vee \cdots \vee L_m \vee D'$, where (1) $m \geq 1$, (2) the literals L_i are of opposite polarity to L, (3) L's atom is $\mathsf{p}(\vec{s}_n)$, (4) L_i's atom is $\mathsf{p}(\vec{t_i})$ for each i, and (5) C and D have no variables in common. The clause $\left(\bigvee_{i=1}^{m} \bigvee_{j=1}^{n} s_j \not\approx t_{ij}\right) \vee C' \vee D'$ is a *flat L-resolvent* of C and D.

Definition 21: A clause $C = L \vee C'$ 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 unsatisfiability [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 22: Consider the clause set N consisting of $C = \mathsf{p}(x,x)$ and $D = \neg\,\mathsf{p}(y_1,y_3) \vee \mathsf{p}(y_1,y_2) \vee \mathsf{p}(y_2,y_3)$. Note that if no literal is selected, all literals are eligible for superposition. In particular, the superposition of $\mathsf{p}(x,x)$ into D's negative literal eventually needs to be performed regardless of the chosen selection function or term order, with the conclusion $E_1 = \mathsf{p}(z_1,z_2) \vee \mathsf{p}(z_2,z_1)$. Then, superposition of E_1 into D yields $E_2 = \mathsf{p}(z_1,z_2) \vee \mathsf{p}(z_2,z_3) \vee \mathsf{p}(z_3,z_1)$. Repeating this process yields infinitely many clauses $E_i = \mathsf{p}(z_1,z_2) \vee \cdots \vee \mathsf{p}(z_i,z_{i+1}) \vee \mathsf{p}(z_{i+1},z_1)$ 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 23: Consider the unsatisfiable set $N = \{C_1, \ldots, C_6\}$, where

$$C_1 = \neg\,\mathsf{c} \vee \mathsf{e} \vee \boxed{\neg\,\mathsf{a}} \qquad C_2 = \neg\,\mathsf{c} \vee \boxed{\neg\,\mathsf{e}} \qquad C_3 = \mathsf{b} \vee \boxed{\mathsf{c}}$$
$$C_4 = \neg\,\mathsf{b} \vee \boxed{\neg\,\mathsf{c}} \qquad C_5 = \mathsf{a} \vee \boxed{\mathsf{b}} \qquad C_6 = \mathsf{c} \vee \boxed{\neg\,\mathsf{b}}$$

Assume the simplification ordering $\mathsf{a} \prec \mathsf{b} \prec \mathsf{c} \prec \mathsf{d} \prec \mathsf{e}$ 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 $C_7 = \mathsf{b} \vee \boxed{\neg\,\mathsf{b}}$, and from C_5 into C_6, yielding $C_8 = \mathsf{a} \vee \boxed{\mathsf{c}}$. Clause C_7 is clearly redundant, whereas C_8 is blocked by its first literal. If we allow removing blocked clauses, the prover enters a loop: C_8 is repeatedly generated and deleted. Thus, the prover will never generate the empty clause for this unsatisfiable set.

As with hidden tautologies, removing blocked clauses breaks the invariant of the given clause procedure that all inferences between clauses in $\mathcal{A}$ are redundant. To see this, assume the setting of Example 23, and let $\mathcal{P} = N$ and $\mathcal{A} = \emptyset$. Assume C_1, C_2, C_3 are moved to the active set. As there are no possible inferences between them, the proof state becomes $\mathcal{A} = \{C_1, C_2, C_3\}$ and $\mathcal{P} = \{C_4, C_5, C_6\}$. After C_4 is moved to $\mathcal{A}$, the conclusion C_7 is computed, but it is not added to $\mathcal{P}$ as it is redundant. Moving C_5 to $\mathcal{A}$ produces no new conclusions, but after C_6 is moved, C_8 is produced. However, if we allow eliminating blocked clauses, it will not be added to $\mathcal{P}$ as it is blocked. The prover then terminates with $\mathcal{A} = N$ and $\mathcal{P} = \emptyset$, even though the original set N is unsatisfiable.

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 22), 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 [29] 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 [30, Sect. 5.4].

Definition 24: A literal L is a *global asymmetric literal* (GAL) for a clause C and a clause set N if for every ground instance $C\sigma$ of C, there exists a ground instance $D\varrho \vee L'\varrho$ of $D \vee L' \in N \setminus \{C\}$ such that $D\varrho \subseteq C\sigma$ and $\neg L'\varrho = L\sigma$.

Every asymmetric literal is GAL, but the converse does not hold:

Example 25: Consider a clause $C = \mathsf{p}(x,y)$ and a clause set $N = \{\mathsf{q} \vee \mathsf{p}(\mathsf{a},\mathsf{a})\}$. Then, $\neg\mathsf{q}$ is not an asymmetric literal for C and N, but it is a GAL for C and N.

Adding and removing GALs preserves and reflects satisfiability:

Theorem 26: If L is a GAL for the clause C and the clause set N, then the set $(N \setminus \{C\}) \cup \{C \vee L\}$ is satisfiable if and only if N is satisfiable.

For first-order logic without equality, a clause $L \vee C$ is blocked if all its L-resolvents are tautologies [16]. The L-resolvent between $L \vee C$ and $\neg L_1 \vee \cdots \vee \neg L_n \vee D$ is $(C \vee D)\sigma$, where σ is the most general unifier of the literals $L, L_1, \ldots, L_n$ [21]. Given a Herbrand model $\mathcal{J}$ of a problem, the following procedure removes all clauses while preserving satisfiability:

1) Let q be a fresh predicate symbol. For each atom $\mathsf{p}(\vec{s})$ in the Herbrand universe: If $\mathcal{J} \models \mathsf{p}(\vec{s})$, add the clause $\mathsf{q} \vee \mathsf{p}(\vec{s})$; otherwise, add $\mathsf{q} \vee \neg\mathsf{p}(\vec{s})$. Adding either clause preserves satisfiability as both are blocked by q.
2) Since $\mathcal{J}$ is a model, for each ground instance $C\sigma$, there exists a clause $\mathsf{q} \vee L$ with $L \in C\sigma$. We can transform $C \in N$ into $C \vee \neg\mathsf{q}$, since $\neg\mathsf{q}$ is a GAL for C and N.
3) Consider the clause $\mathsf{q} \vee L$ added by step 1. Since L is ground and no clause $\mathsf{q} \vee \neg L$ was added (since $\mathcal{J}$ is a model), the only L-resolvents are against clauses added by step 2. Since all of those clauses contain $\neg\mathsf{q}$, the resolvents are tautologies. Thus, each $\mathsf{q} \vee L$ is blocked and can be removed in turn.
4) The remaining clauses all contain the literal $\neg\mathsf{q}$. 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 $\mathsf{q} \vee L$ [16].) The procedure also terminates only for finite Herbrand models.

Example 27: Consider the satisfiable clause set $N = \{\mathsf{r}(x) \vee \mathsf{s}(x), \neg\mathsf{r}(\mathsf{a}), \neg\mathsf{s}(\mathsf{b})\}$ and a Herbrand model $\mathcal{J}$ over $\{\mathsf{a},\mathsf{b},\mathsf{r},\mathsf{s}\}$ such that $\mathsf{r}(\mathsf{b})$ and $\mathsf{s}(\mathsf{a})$ are the only true atoms in $\mathcal{J}$. We show how to remove all clauses in N using $\mathcal{J}$ by following the procedure above.

Let $N_{\mathcal{J}} = \{\mathsf{q} \vee \neg\mathsf{r}(\mathsf{a}), \mathsf{q} \vee \mathsf{r}(\mathsf{b}), \mathsf{q} \vee \mathsf{s}(\mathsf{a}), \mathsf{q} \vee \neg\mathsf{s}(\mathsf{b})\}$. We set $N \leftarrow N \cup N_{\mathcal{J}}$. This preserves satisfiability since all clauses in $N_{\mathcal{J}}$ are blocked. It is easy to check that $\neg\mathsf{q}$ is GAL for every clause in $N \setminus N_{\mathcal{J}}$. The only substitutions that need to be considered are $\{x \mapsto \mathsf{a}\}$ and $\{x \mapsto \mathsf{b}\}$ for $\mathsf{r}(x) \vee \mathsf{s}(x)$. So we set $N \leftarrow \{\neg\mathsf{q} \vee \mathsf{r}(x) \vee \mathsf{s}(x), \neg\mathsf{q} \vee \neg\mathsf{r}(\mathsf{a}), \neg\mathsf{q} \vee \neg\mathsf{s}(\mathsf{b})\} \cup N_{\mathcal{J}}$. Clearly, all clauses in $N_{\mathcal{J}}$ are blocked, so we set $N \leftarrow N \setminus N_{\mathcal{J}}$. All clauses remaining in N have a literal $\neg\mathsf{q}$ 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 Kiesl et al. [16]. All techniques are implemented in the Zipperposition prover [31]. Zipperposition is designed for fast prototyping of improvements to superposition, but it implements many of the most successful heuristics from the E prover [32] and has recently become quite competitive [33].

A. Hidden-Literal-Based Elimination

For HLBE, an efficient representation of $\mathrm{HL}(L,N)$ 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 $\mathit{Imp}[L']$ associates with each literal L' a set of pairs (L,M) such that $L' \hookrightarrow^k L$, where $k \leq K_{\mathrm{len}}$ and M is the multiset of clauses used to derive $L' \hookrightarrow^k L$. Moreover, we consider only transitions of type (1) (as per Definition 4). The following algorithm maintains Imp dynamically, updating it as the prover derives and deletes clauses. It depends on the global variable Imp and the parameters K_{len} and K_{imp}.

```
procedure AddImplication($L_a, L_c, C$)
  if $Imp[L_a\sigma] \neq \emptyset$ for some renaming $\sigma$ then
    $(L_a, L_c) \leftarrow (L_a\sigma, L_c\sigma)$
  if there are no $L, L', M, \sigma$ such that $(L', M) \in Imp[L]$,
      $L\sigma = L_a$, and $L'\sigma = L_c$ then
    for all $(\sigma, M)$ such that $(L_c\sigma, M) \in Imp[L_a\sigma]$ do
      erase all $(L', M')$ such that $M \subseteq M'$ from $Imp[L_a\sigma]$
    for all $L$ such that $(L', M) \in Imp[L]$
```

```
            and $L_a\sigma = L'$ for some $\sigma$ do
        if $|M| < K_{len}$ then
          $Imp[L] \leftarrow Imp[L] \cup \{(L_c\sigma, M \uplus \{C\})\}$
      for all $L$ such that $Imp[L] \neq \emptyset$
            and $L\sigma = L_c$ for some $\sigma$ do
        $Concl \leftarrow \{(L'\sigma, M \uplus \{C\}) \mid$
                  $(L', M) \in Imp[L], |M| < K_{len}\}$
        $Imp[L_a] \leftarrow Imp[L_a] \cup Concl$
      $Congr \leftarrow \{(s \not\approx t, \{C\}) \mid \exists u.\, L_c = u[s] \not\approx u[t]\}$
      $Imp[L_a] \leftarrow Imp[L_a] \cup \{(L_c, \{C\})\} \cup Congr$

  procedure TrackClause($C$)
    if $C = L_1 \vee L_2$ then
      AddImplication($\neg L_1$, $L_2$, $C$)
      AddImplication($\neg L_2$, $L_1$, $C$)
      if $L_2 = \neg L_1\sigma$ for some nonidempotent $\sigma$ then
        for all $i \leftarrow 1$ to $K_{imp}$ do
          $L_2 \leftarrow L_2\sigma$
          AddImplication($\neg L_1$, $L_2$, $C$)

  procedure UntrackClause($C$)
    for all $L_a, L_c, M$ such that $(L_c, M) \in Imp[L_a]$ do
      if $C \in M$ then
        erase $(L_c, M)$ from $Imp[L_a]$
```

The algorithm views a clause $L \vee L'$ as two implications $\neg L \longrightarrow L'$ and $\neg L' \longrightarrow L$. It stores only one entry for all literals equal up to variable renaming (line 2). Each implication $L_a \longrightarrow L_c$ 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 $L_a \longrightarrow L_c$: Either L_c becomes the new consequent (line 9) or L_a becomes the new antecedent (line 13). If L_c can be decomposed into $u[s] \not\approx u[t]$, rule (3) of Definition 4 allows us to store $s \not\approx t$ in $Imp[L_a]$ (line 18). This is an exception to the idea that transitive chains should use only rule (1). The application of rule (3) does not count toward the bound K_{len}. If L_a is of the form $u[s] \approx u[t]$, then *Imp* could be extended so that $Imp[s \approx t] = Imp[L_a]$, but this would substantially increase *Imp*'s memory footprint.

In first-order logic, different instances of the same clause can be used along a transitive chain. For example, the clause $C = \neg\, \mathsf{p}(x) \vee \mathsf{p}(\mathsf{f}(x))$ induces $\mathsf{p}(x) \hookrightarrow^i \mathsf{p}(\mathsf{f}^i(x))$ for all i. The algorithm discovers such self-implications (line 23): For each clause C of the form $\neg L \vee L\sigma$, where σ is some nonidempotent substitution, the entires $(L\sigma^2, \{C\}), \ldots, (L\sigma^{K_{imp}+1}, \{C\})$ 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 [34]. Given a query literal L, this indexing data structure efficiently finds all literals L' such that for some σ, $L'\sigma = L$ 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., $Imp^{-1}[L] = \{L' \mid Imp[L'] = (L, M) \text{ for some } M\}$. 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 $Imp[L]$. Finally, we limit the number of entries stored in $Imp[L]$ – by default, up to 48 pairs in each $Imp[L]$ are stored.

Rules HLE and HTR have a simple implementation based on *Imp* lookups. To implement UnitHLE and UnitHTR, we maintain the index *Unit*, containing literals $L_c\sigma$, such that $(L_c, M) \in Imp[L_a]$ for some M and L_a and σ is the most general unifier of L' and L_a, for some unit clause $\{L'\}$. The 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 conventional approach constructs the *binary implication graph* for the clause set N [4], with edges $(\neg L, L')$ and $(\neg L', L)$ whenever $L \vee L' \in N$. To avoid traversing the graph repeatedly, solvers rely on timestamps to discover connections between literals. This relies on syntactic literal comparisons, which is very fast 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 the prover emits whenever a clause is added to or removed from the proof state and update the records. At the beginning of the 1st, $(K_{iter}+1)$st, $(2K_{iter}+1)$st, $\ldots$ 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.

The most important novel aspect of our predicate elimination implementation is recognizing the definition clauses for symbol p in a clause set N, which is performed as follows:

1) Let $G = \{C \mid C = (\neg)\, \mathsf{p}(\vec{x}) \vee C', C \in N$, no variable repeats in $\vec{x}$, and variables of C' are among $\vec{x}\}$. If G is empty, report failure; otherwise continue.
2) Rename all clauses in G so that their only variables are $\vec{x}$.
3) Let $\lfloor a \rfloor$ be a function that assigns a propositional variable to each atom a. This function is lifted to literals by assigning $\lfloor \neg a \rfloor = \neg x$, if $\lfloor a \rfloor = x$, and to clauses pointwise. Furthermore, let $E = \{\lfloor C' \rfloor \mid (\neg)\, \mathsf{p}(\vec{x}) \vee C' \in G\}$. If E is satisfiable, report failure. Else, let E' be the unsatisfiable

core of E and G' the set of corresponding first-order clauses and continue.

4) If all resolvents in $G'_{\mathsf{p}} \bowtie_{\mathsf{p}} G'_{\neg\mathsf{p}}$ are tautologies, then G' is the definition set for symbol p. Else, report failure.

The invalidity of set E from step 3 is checked using a SAT solver, which is already integrated in Zipperposition. As modern theorem provers (such as E or Vampire) also use SAT solvers, the method can easily be implemented.

During experimentation, we noticed that recognizing definitions of symbols that occur in the conjecture often harms performance. Thus, Zipperposition recognizes definitions only for the remaining 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 [35] 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 clauses from the active set $\mathcal{A}$, only clauses from the passive set $\mathcal{P}$, 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 $K_{\text{len}} = 2$ and $K_{\text{imp}} = 0$.The columns specify different limits on the number of tracked clauses, with ∞ denoting that no limit is imposed. The rows represent different kinds of tracked clauses. 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

	Tracked clauses			
	250	500	1000	∞
Active	−14	−16	−8	−12
Passive	+7	+10	+5	−35
Both	**+12**	+10	+7	−45

Fig. 1. Impact of the number and kinds of tracked clauses on HLBE performance, when only predicate literals are tracked

	Tracked clauses			
	250	500	1000	∞
Active	−10	−14	−8	−18
Passive	−5	−5	−14	−71
Both	**+2**	−1	−8	−79

Fig. 2. Impact of the number and kinds of tracked clauses on HLBE performance, when all literals are tracked

DISCOUNT-loop provers [26] 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 $K_{\text{tol}} > 0$, 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 $K_{\text{tol}} = 0$, SPE proves 60 theorems not proved using $K_{\text{tol}} = 50$. The effect weakens as K_{tol} grows. When $K_{\text{tol}} = 100$, SPE proves only 13 problems not found when $K_{\text{tol}} = 200$. Similarly, the set of problems proved by SPE and PPE differs: When $K_{\text{tol}} = 25$, 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 $K_{\text{tol}} = 25$ and HLBE which tracks up to 500 clauses from any clause set, $K_{\text{len}} = 2$, and $K_{\text{imp}} = 0$. We use each technique as preprocessing and inprocessing.

	Chain length K_{len} 1	2	4	8
$K_{\text{imp}} = 0$	$+9$	$+10$	$+7$	$+5$
$K_{\text{imp}} = 1$	$+5$	$+$**11**	$+7$	$+4$
$K_{\text{imp}} = 2$	$+6$	$+$**11**	$+8$	$+8$

Fig. 3. Impact of the parameters K_{len} and K_{imp} on HLBE performance

	K&K	Relaxed with K_{tol} 0	25	50	100	200
SPE preproc.	$+70$	$+117$	$+154$	$+160$	$+154$	$+158$
PPE preproc.	$+71$	$+124$	$+160$	$+164$	$+$**165**	$+162$

Fig. 4. Impact of the choice of criterion on predicate elimination performance

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 from our benchmark set, increasing the success rate by roughly 0.4%. The same percentage increase was obtained Kiesl et al. Using BCE as 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.

	BCE	SPE	SPE +BCE	PPE	PPE +BCE	HLBE +PPE +BCE
Preprocessing	$+30$	$+154$	$+159$	$+160$	$+$**166**	$+162$
Inprocessing	-48	$+140$	$+127$	$+146$	$+131$	$+127$

Fig. 5. Performance of predicate and blocked clause elimination

	BCE	SPE	SPE +BCE	PPE	PPE +BCE	HLBE +PPE +BCE
Preprocessing	$+29$	$+46$	$+$**60**	$+47$	$+59$	$+55$

Fig. 6. Performance of predicate and blocked clause elimination for establishing satisfiability

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 [36] and blocked clause addition [37]. Sixth, the techniques we covered could be adapted to work with other first-order calculi, or generalized further to work with higher-order calculi such as combinatory superposition [38] and λ-superposition [39].

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 23 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

[1] J. P. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 185, pp. 131–153.

[2] L. Bachmair and H. Ganzinger, "Rewrite-based equational theorem proving with selection and simplification," *J. Log. Comput.*, vol. 4, no. 3, pp. 217–247, 1994.

[3] M. J. H. Heule, M. Järvisalo, and A. Biere, "Clause elimination procedures for CNF formulas," in *LPAR-17*, ser. LNCS, C. G. Fermüller and A. Voronkov, Eds., vol. 6397. Springer, 2010, pp. 357–371.

[4] ——, "Efficient CNF simplification based on binary implication graphs," in *SAT 2011*, ser. LNCS, K. A. Sakallah and L. Simon, Eds., vol. 6695. Springer, 2011, pp. 201–215.

[5] J. W. Freeman, "Improvements to propositional satisfiability search algorithms," Ph.D. dissertation, University of Pennsylvania, 1995.

[6] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[7] S. Subbarayan and D. K. Pradhan, "NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances," in *SAT 2004*, ser. LNCS, H. H. Hoos and D. G. Mitchell, Eds., vol. 3542. Springer, 2004, pp. 276–291.

[8] P. Chatalic and L. Simon, "ZRES: The old Davis–Putnam procedure meets ZBDD," in *CADE-18*, ser. LNCS, D. A. McAllester, Ed., vol. 1831. Springer, 2000, pp. 449–454.

[9] A. Biere, "Resolve and expand," in *SAT 2004*, ser. LNCS, H. H. Hoos and D. G. Mitchell, Eds., vol. 3542. Springer, 2004, pp. 59–70.

[10] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *SAT 2005*, ser. LNCS, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75.

[11] D. M. Gabbay and H. J. Ohlbach, "Quantifier elimination in second-order predicate logic," in *KR '92*, B. Nebel, C. Rich, and W. R. Swartout, Eds. Morgan Kaufmann, 1992, pp. 425–435.

[12] H. J. Ohlbach, "SCAN—elimination of predicate quantifiers," in *CADE-13*, ser. LNCS, M. A. McRobbie and J. K. Slaney, Eds., vol. 1104. Springer, 1996, pp. 161–165.

[13] Z. Khasidashvili and K. Korovin, "Predicate elimination for preprocessing in first-order theorem proving," in *SAT 2016*, ser. LNCS, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 361–372.

[14] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *TACAS 2010*, ser. LNCS, J. Esparza and R. Majumdar, Eds., vol. 6015. Springer, 2010, pp. 129–144.

[15] A. Biere, F. Lonsing, and M. Seidl, "Blocked clause elimination for QBF," in *CADE-23*, ser. LNCS, N. Bjørner and V. Sofronie-Stokkermans, Eds., vol. 6803. Springer, 2011, pp. 101–115.

[16] B. Kiesl, M. Suda, M. Seidl, H. Tompits, and A. Biere, "Blocked clauses in first-order logic," in *LPAR-21*, ser. EPiC Series in Computing, T. Eiter and D. Sands, Eds., vol. 46. EasyChair, 2017, pp. 31–48.

[17] G. Sutcliffe, "The TPTP problem library and associated infrastructure—from CNF to TH0, TPTP v6.4.0," *J. Autom. Reason.*, vol. 59, no. 4, pp. 483–502, 2017.

[18] P. Vukmirović, J. Blanchette, and M. J. H. Heule, "SAT-inspired eliminations for superposition (technical report)," Technical report, 2021, https://matryoshka-project.github.io/pubs/satelimsup_report.pdf.

[26] J. Avenhaus, J. Denzinger, and M. Fuchs, "DISCOUNT: A system for distributed equational deduction," in *RTA-95*, ser. LNCS, J. Hsiang, Ed., vol. 914. Springer, 1995, pp. 397–402.

[19] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Wiley, 1987.

[20] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.

[21] L. Bachmair and H. Ganzinger, "Resolution theorem proving," in *Handbook of Automated Reasoning*, J. A. Robinson and A. Voronkov, Eds. Elsevier and MIT Press, 2001, vol. I, pp. 19–99.

[22] A. Riazanov and A. Voronkov, "Splitting without backtracking," in *IJCAI 2001*, B. Nebel, Ed. Morgan Kaufmann, 2001, pp. 611–617.

[23] A. Fietzke and C. Weidenbach, "Labelled splitting," *Ann. Math. Artif. Intell.*, vol. 55, no. 1–2, pp. 3–34, 2009.

[24] A. Voronkov, "AVATAR: The architecture for first-order theorem provers," in *CAV 2014*, ser. LNCS, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 696–710.

[25] W. McCune and L. Wos, "Otter—the CADE-13 competition incarnations," *J. Autom. Reason.*, vol. 18, no. 2, pp. 211–220, 1997.

[27] A. Nonnengart and C. Weidenbach, "Computing small clause normal forms," in *Handbook of Automated Reasoning*, J. A. Robinson and A. Voronkov, Eds. Elsevier and MIT Press, 2001, vol. I, pp. 335–367.

[28] M. Heule, M. Seidl, and A. Biere, "A unified proof system for QBF preprocessing," in *IJCAR 2014*, ser. LNCS, S. Demri, D. Kapur, and C. Weidenbach, Eds., vol. 8562. Springer, 2014, pp. 91–106.

[29] B. Kiesl and M. Suda, "A unifying principle for clause elimination in first-order logic," in *CADE-26*, ser. LNCS, L. de Moura, Ed., vol. 10395. Springer, 2017, pp. 274–290.

[30] M. Fitting, *First-Order Logic and Automated Theorem Proving*, 2nd ed., ser. Graduate Texts in Computer Science. Springer, 1996.

[31] S. Cruanes, "Superposition with structural induction," in *FroCoS 2017*, ser. LNCS, C. Dixon and M. Finger, Eds., vol. 10483. Springer, 2017, pp. 172–188.

[32] S. Schulz, S. Cruanes, and P. Vukmirović, "Faster, higher, stronger: E 2.3," in *CADE-27*, ser. LNCS, P. Fontaine, Ed., vol. 11716. Springer, 2019, pp. 495–507.

[33] G. Sutcliffe, "The CADE-27 Automated Theorem Proving System Competition—CASC-27," *AI Commun.*, vol. 32, no. 5-6, pp. 373–389, 2020.

[34] I. V. Ramakrishnan, R. C. Sekar, and A. Voronkov, "Term indexing," in *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001, vol. II, pp. 1853–1964.

[35] A. Stump, G. Sutcliffe, and C. Tinelli, "StarExec: A cross-community infrastructure for logic solving," in *IJCAR 2014*, ser. LNCS, S. Demri, D. Kapur, and C. Weidenbach, Eds., vol. 8562. Springer, 2014, pp. 367–373.

[36] N. Manthey, M. Heule, and A. Biere, "Automated reencoding of Boolean formulas," in *HVC 2012*, ser. LNCS, A. Biere, A. Nahir, and T. E. J. Vos, Eds., vol. 7857. Springer, 2012, pp. 102–117.

[37] O. Kullmann, "On a generalization of extended resolution," *Discr. Appl. Math.*, vol. 96–97, pp. 149–176, 1999.

[38] A. Bhayat and G. Reger, "A combinator-based superposition calculus for higher-order logic," in *IJCAR 2020, Part I*, ser. LNCS, N. Peltier and V. Sofronie-Stokkermans, Eds., vol. 12166. Springer, 2020, pp. 278–296.

[39] A. Bentkamp, J. Blanchette, S. Tourret, P. Vukmirović, and U. Waldmann, "Superposition with lambdas," *J. Autom. Reasoning*, vol. 65, no. 7, pp. 893–940, 2021.