SAT-INSPIRED HIGHER-ORDER ELIMINATIONS

JASMIN BLANCHETTE [a,b] AND PETAR VUKMIROVIĆ [a]

[a] Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands
e-mail address: j.c.blanchette@vu.nl
e-mail address: petar.vukmirovic2@gmail.com

[b] Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany
e-mail address: jasmin.blanchette@mpi-inf.mpg.de

Abstract. We generalize several propositional preprocessing techniques to higher-order logic, building on existing first-order generalizations. These techniques eliminate literals, clauses, or predicate symbols from the problem, with the aim of making it more amenable to automatic proof search. We also introduce a new technique, which we call *quasipure literal elimination*, that strictly subsumes pure literal elimination. The new techniques are implemented in the Zipperposition theorem prover. Our evaluation shows that they sometimes help prove problems originating from the TPTP library and Isabelle formalizations.

1. Introduction

Processing techniques are an important optimization in SAT (Boolean satisfiability) solving. Following up on early work in the 1990s [GO92, Ohl96], there has recently been renewed interest in adapting propositional techniques to first-order logic [KK16, KSS+17, VBH21a], resulting in a noticeable increase of the success rate of automatic theorem provers based on superposition [BG94].

In this article, we consider the extension of four main classes of SAT preprocessing techniques to classical higher-order logic. These extensions are called hidden-literal-based elimination (Section 3), predicate elimination (Section 4), blocked clause elimination (Section 5), and quasipure literal elimination (Section 6). Elimination techniques make the problem simpler and hence possibly more amenable to automatic proof search. The techniques can be used either to preprocess the problem or to transform the prover's current clause set during proof search, a use that is sometimes called *inprocessing*.

An advantage of preprocessing is its greater generality: Preprocessing techniques can be used in tandem with any higher-order proof calculus, as long as the calculus is built around a notion of clause. We assume that a clausifier is run as a preprocessor and introduces some clausal structure. The more clausal structure it produces, the more effective the elimination techniques can be. Examples of provers compatible with the techniques are λE [VBS], Leo-III [SB18], Vampire [BR20], and Zipperposition [BBTV21].

Our setting is a clausal version of classical rank-1 polymorphic higher-order logic (Section 2). Since previous work focused on an untyped or monomorphic logic, our work also

Preprint submitted to
Logical Methods in Computer Science

generalizes this aspect. Higher-order logic also distinguishes between standard and general (Henkin) semantics. Since calculi are proved complete with respect to general semantics, this is the semantics we adopt. Our techniques preserve the unsatisfiability of problems, and therefore their provability by a complete prover. In addition, they preserve the satisfiability of problems, and therefore their unprovability by a sound prover.

The main difficulty we face in higher-order logic concerns predicate elimination and blocked clause elimination, which are both based on resolution. In first-order logic, a literal $\neg\, \mathsf{p}(\vec{s})$ can only be resolved against a literal $\mathsf{p}(\vec{t})$, with the same predicate symbol p. By contrast, in higher-order logic, $\neg\, \mathsf{p}\ \vec{s}$ can be resolved against any variable-headed literal $y\ \vec{t}$, for example by taking $y := \lambda\vec{x}.\, \mathsf{p}\ \vec{s}$, where the bound variables $\vec{x}$ are fresh. We will see that this issue can be circumvented: A key finding of this article is that we can ignore variable-headed resolvents and focus on the p-literals.

Another potential issue is that higher-order logic can have infinitely many resolvents. For example, resolving $\neg\, \mathsf{p}\, (\mathsf{f}\, (y\, \mathsf{a}))$ and $\mathsf{p}\, (y\, (\mathsf{f}\, \mathsf{a}))$ produces infinitely many conclusions of the form $\mathsf{p}\, (\mathsf{f}\, (\ldots (\mathsf{f}\, \mathsf{a}) \ldots))$. Again, the issue is not as severe as it looks, because the variant of resolution we use—flat resolution—does not unify terms.

Example 1.1. To give a flavor of our elimination techniques, let us review an example involving blocked clause elimination. Let $\mathsf{a} : \iota$, $\mathsf{p} : \iota \to o$, and $\mathsf{choice} : (\iota \to o) \to \iota$ be symbols, where o is the type of Booleans and ι is a base type. Consider the clause set

$$N = \{\neg\, y\, z \vee y\, (\mathsf{choice}\, y),\ \mathsf{q}\, \mathsf{a},\ \neg\, \mathsf{q}\, (\mathsf{choice}\, \mathsf{q}),\ \mathsf{p}\, \mathsf{a},\ \neg\, \mathsf{p}\, z \vee z \approx \mathsf{a}\}$$

The set is clearly inconsistent because the $\{y \mapsto \mathsf{q},\ z \mapsto \mathsf{a}\}$ instance of the first clause is inconsistent with the second and third clauses.

Under some basic conditions, a clause C containing a literal L is said to be "blocked" if all of its so-called binary flat L-resolvents with clauses from $N \setminus \{C\}$ are tautologies. (We will see in Section 5 what this means exactly.) The fourth clause is blocked by its literal $\mathsf{p}\, \mathsf{a}$ because its only binary flat $(\mathsf{p}\, \mathsf{a})$-resolvent, with the fifth clause, is the tautology $\mathsf{a} \not\approx z \vee z \approx \mathsf{a}$. Similarly, the fifth clause is blocked by its literal $\neg\, \mathsf{p}\, z$ because its only binary flat $(\neg\, \mathsf{p}\, z)$-resolvent, with the fourth clause, is the same tautology. Either or both clauses can be removed without from N losing unsatisfiability.

All the techniques are implemented in the higher-order prover Zipperposition (Section 7), allowing us to measure their effectiveness on benchmarks originating from the TPTP library [Sut17] and Isabelle [NPW02] formalizations (Section 8). The raw experimental data are available online.[1]

We will mention closely related research in the relevant sections. We point to Vukmirović et al. [VBH21a] for a more detailed discussion of related work.

2. Clausal Higher-Order Logic

The logic we use as a basis of our work is a rank-1 polymorphic higher-order logic with general semantics and both functional and Boolean extensionality. It corresponds essentially to the logic embodied by the TPTP TH1 format [KSR16], including Hilbert choice. Our conventions largely follow those used by Bentkamp, Blanchette, Tourret, and Vukmirović to define λ-superposition [BBTV21]. Our presentation is based on theirs.

[1] https://zenodo.org/record/6997515

In higher-order logic, formulas are simply terms of Boolean type. Briefly, our version of the logic also has a clausal outer structure, as found in several higher-order provers. Clauses are then built around terms as an extra layer of structure. We write formula-level Boolean operators in bold (e.g., $\boldsymbol{\neg}$, $\boldsymbol{\vee}$, $\boldsymbol{\forall}$) to distinguish them from clause-level operators.

2.1. **Syntax.** Let us define the syntax of our logic more precisely, starting with types. Throughout this article, we use the notation $\vec{a}_n$, or simply $\vec{a}$, to denote an n-tuple $(a_1, \ldots, a_n)$. Sometimes we might also write $\vec{a}_i$, meaning $(a_{i1}, \ldots, a_{in})$, to be distinguished from $\vec{a}_i = (a_1, \ldots, a_i)$.

We start by fixing an infinite set $\mathcal{V}_{\mathsf{ty}}$ of type variables. A set Σ_{ty} of type constructors with associated arities is a *type signature*. We require the presence of a nullary type constructor o, for Booleans, and a binary type constructor $\to$ for functions. We let α range over type variables and κ over type constructors. A *type*, ranged over by τ and υ, is defined inductively to be either a variable $\alpha \in \mathcal{V}_{\mathsf{ty}}$ or an expression $\kappa(\vec{\tau}_n)$, where κ is an n-ary type constructor and $\vec{\tau}$ in an n-tuple of types. If $n = 0$, we write κ instead of $\kappa()$. In addition, expressions with $\kappa = \to$ are written in infix notation, as $\tau_1 \to \tau_2$. A *type declaration* is an expression $\Pi\vec{\alpha}_m.\, \tau$, where $\vec{\alpha}$ consists of distinct type variables and all the type variables occurring in τ belong to $\vec{\alpha}$. If $m = 0$, we write τ instead of $\Pi.\, \tau$.

Next, we fix a type signature Σ_{ty} and a set $\mathcal{V}$ of term variables with associated types. We require that there are infinitely many variables of each type. A *term signature* is a set Σ of *symbols* $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{f}, \mathsf{p}, \mathsf{q}, \ldots$, each associated with a type declaration. (Often, symbols are called "constants" in the higher-order logic literature.) A symbol with type declaration $\Pi\vec{\alpha}.\, \tau_1 \to \cdots \to \tau_n \to o$ is called a predicate symbol. We write $\mathsf{f} : \Pi\vec{\alpha}.\, \tau$ to indicate that symbol f has type signature $\Pi\vec{\alpha}.\, \tau$. We require the presence in Σ of the logical symbols $\boldsymbol{\bot}, \boldsymbol{\top}, \boldsymbol{\neg}, \boldsymbol{\wedge}, \boldsymbol{\vee}, \boldsymbol{\to}, \boldsymbol{\forall}, \boldsymbol{\exists}, \boldsymbol{\approx}, \boldsymbol{\not\approx}$ with their usual type declarations (e.g., $\Pi\alpha.\, \alpha \to \alpha \to o$ for $\boldsymbol{\approx}$). We also assume the presence of the Hilbert choice operator $\boldsymbol{\varepsilon} : \Pi\alpha.\, (\alpha \to o) \to \alpha$. We will generally leave the signature implicit, assuming some fixed signature Σ.

We now introduce three notions of terms: raw λ-terms, λ-terms, and actual terms. The λ-terms are α-equivalence classes of raw λ-terms, and the actual terms are $\beta\eta$-equivalence classes of λ-terms.

More precisely, the *raw λ-terms* are defined inductively as follows:

- Every variable x of type τ is a raw λ-term of type τ.
- If f has type declaration $\Pi\vec{\alpha}_m.\, \tau$ in Σ and $\vec{\upsilon}_m$ is a tuple of types, called *type arguments*, then $\mathsf{f}\langle\vec{\upsilon}_m\rangle$ is a raw λ-term of type $\tau\{\vec{\alpha}_m \mapsto \vec{\upsilon}_m\}$. If $m = 0$, we simply write f instead of $\mathsf{f}\langle\rangle$.
- If x is a variable of type τ and t is a term of type υ, then the *λ-abstraction* $\lambda x.\, t$ is a raw λ-term of type $\tau \to \upsilon$.
- If s is a term of type $\tau \to \upsilon$ and t is a term of type τ, then the *application* $s\, t$ is a raw λ-term of type υ.

We abbreviate $\lambda x_1.\, \ldots\, \lambda x_n.\, t$ to $\lambda x_1\, \ldots\, x_n.\, t$ or $\lambda\vec{x}_n.\, t$, $\boldsymbol{\forall}\,(\lambda x.\, t)$ to $\boldsymbol{\forall} x.\, t$, and similarly for $\boldsymbol{\exists}$. Abusing notation, we also write $t\, \vec{u}_n$ for $t\, u_1\, \ldots\, u_n$. We assume standard notions of free and bound variables as well as subterms. To indicate that a term t has a type τ, we write $t : \tau$.

The α-renaming rule of the λ-calculus relates two raw λ-terms if the two are equal up to (capture-avoiding) renaming of their bound variables. For example, $\lambda x.\, \mathsf{f}\, x\, x$ and $\lambda y.\, \mathsf{f}\, y\, y$ are α-renamings of each other. Two raw λ-terms are α-equivalent if they can be made equal by

α-renaming their subterms. The *λ-terms* consist of the equivalence classes of raw λ-terms modulo α-equivalence of subterms. We assume the standard notion of (capture-avoiding) substitution on λ-terms. We also define a notion of replacement: $t[\mathsf{f} \mapsto u]$ denotes the term obtained by replacing all occurrences of f in t with a term u of the same type.

The β-reduction rule relates two λ-terms if the first one has the form $(\lambda x.\, s)\, t$ and the second one has the form $s\{x \mapsto t\}$, where bound variables in s are implicitly renamed to avoid capture. The η-reduction rule relates two λ-terms if the first one has the form $\lambda x.\, t\, x$ and the second one has the form t, where t contains no free occurrences of x. For example, $(\lambda x.\, \mathsf{f}\, x\, x)\, \mathsf{b}$ β-reduces to $\mathsf{f}\, \mathsf{b}\, \mathsf{b}$, and $\lambda x.\, \mathsf{f}\, x$ η-reduces to f. Two λ-terms are $\beta\eta$-equivalent if they can be made equal by β- and η-reducing their subterms. The *terms* consist of the equivalence classes of λ-terms modulo $\beta\eta$-equivalence of subterms. The *formulas* are the terms of type o. We let φ, ψ range over formulas.

Convention 2.1. When inspecting the structure of a term, we will consider a representative in η-short β-normal form, obtained by exhaustively applying β- and η-reduction on subterms. Such a representative is unique up to α-equivalence.

An alternative to the η-short β-normal form is the η-long β-normal form, in which unapplied functions are η-expanded rather than η-reduced (i.e., η-reduction is applied in reverse on these). The techniques presented in this article work unchanged in such a setting.

Two terms t, u are *unifiable* if there exists a substitution σ such that $t\sigma = u\sigma$. For example, a and $y\, \mathsf{a}$ are unifiable by taking $y := \lambda x.\, x$. (This works because terms are equal up to β-reduction.) Unification of higher-order terms is undecidable. Types, however, are isomorphic to first-order terms, and hence their unification problem is decidable. Moreover, if a unifier exists, then a most general unifier exists (up to the naming of variables). For example, the most general unifier for the unification problem $\mathsf{pair}(\alpha, \mathsf{nat}) \stackrel{?}{=} \mathsf{pair}(\mathsf{int}, \beta)$ is $\{\alpha \mapsto \mathsf{int}, \beta \mapsto \mathsf{nat}\}$.

Finally, we define literals and clauses on top of terms. An atom is an equation $s \approx t$ corresponding to an unordered pair $\{s, t\}$. (We reserve $=$ for syntactic equality of terms.) A literal is an equation $s \approx t$ or a disequation $s \not\approx t$. Given a predicate symbol p, the literal $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{s} \approx \mathsf{T}$ is abbreviated to $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{s}$, and its complement $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{s} \not\approx \mathsf{T}$ is abbreviated to $\neg\, \mathsf{p}\langle\vec{\alpha}\rangle\ \vec{s}$. Moreover, a p-*literal* is a literal of the form $(\neg)\, \mathsf{p}\langle\vec{\tau}\rangle\ \vec{t}$. Note that it is possible in higher-order logic for a non-p-literal to contain p, or even for the arguments $\vec{t}$ of a p-literal to contain p. Given a literal L, we write $\neg\, L$ to denote its complement, with $\neg\neg\, L = L$.

A clause C is a finite multiset of literals, written as $L_1 \vee \cdots \vee L_n$ and interpreted disjunctively. Clauses are often defined as sets of literals, but multisets are better behaved with respect to substitution: If C has n literals, so has $C\sigma$ regardless of whether σ unifies some of C's literals. The type and term variables contained in a clause are implicitly quantified universally. (Within terms, $\mathsf{\forall}$ and $\mathsf{\exists}$ can be used to quantify over term variables.)

A type, term, or clause is *monomorphic* if it contains no type variables. A term or clause is *closed* if it contains no free term variables.

It is sometimes useful to encode a clause C as a formula. The formula $[C]$ representing the clause C is defined by replacing the nonbold symbols $\approx$, $\not\approx$, and $\vee$ by their bold counterparts $\mathbf{\approx}$, $\mathbf{\not\approx}$, and V. This formula is uniquely defined up to the orientation of the equations and the order of the literals, neither of which affects the semantics.

2.2. **Semantics.** A *type interpretation* $\mathcal{I}_{\mathsf{ty}} = (\mathcal{U}, \mathcal{J}_{\mathsf{ty}})$ consists of two components. The *universe* $\mathcal{U}$ is a collection of nonempty sets, the *domains*. We require $\mathcal{U}$ to contain the domain

$\{0,1\}$, where 0 represents falsehood and 1 represents truth. The function $\mathcal{J}_{\mathsf{ty}}$ associates with each n-ary type constructor κ a function $\mathcal{J}_{\mathsf{ty}}(\kappa) : \mathcal{U}^n \to \mathcal{U}$, with the requirements that $\mathcal{J}_{\mathsf{ty}}(o) = \{0,1\}$ and that the set $\mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{D}_1, \mathcal{D}_2)$ is a subset of the (total) function space from $\mathcal{D}_1$ to $\mathcal{D}_2$ for all domains $\mathcal{D}_1, \mathcal{D}_2 \in \mathcal{U}$. The semantics is *standard* if $\mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{D}_1, \mathcal{D}_2)$ is the entire function space for all $\mathcal{D}_1, \mathcal{D}_2 \in \mathcal{U}$. A *type valuation* ξ is a function that maps every type variable to a domain.

The *denotation* of a type in a type interpretation $\mathcal{I}_{\mathsf{ty}}$ under a type valuation ξ is defined by the recursive equations $[\![\alpha]\!]_{\mathcal{I}_{\mathsf{ty}},\xi} = \xi(\alpha)$ and $[\![\kappa(\vec{\tau})]\!]_{\mathcal{I}_{\mathsf{ty}},\xi} = \mathcal{J}_{\mathsf{ty}}(\kappa)([\![\vec{\tau}]\!]_{\mathcal{I}_{\mathsf{ty}},\xi})$. For monomorphic types τ, the denotation does not depend on the valuation ξ, allowing us to write $[\![\tau]\!]_{\mathcal{I}_{\mathsf{ty}}}$ instead of $[\![\tau]\!]_{\mathcal{I}_{\mathsf{ty}},\xi}$.

A type valuation ξ can be extended to be a *valuation* by additionally assigning an element $\xi(x) \in [\![\tau]\!]_{\mathcal{I}_{\mathsf{ty}},\xi}$ to each variable $x : \tau$. We will sometimes use partial functions as valuations if the values outside the function's domain are irrelevant. An *interpretation function* $\mathcal{J}$ for a type interpretation $\mathcal{I}_{\mathsf{ty}}$ associates with each symbol $\mathsf{f} : \Pi\vec{\alpha}_m.\, \tau$ and domain tuple $\vec{\mathcal{D}} \in \mathcal{U}^m$ a value $\mathcal{J}(\mathsf{f}, \vec{\mathcal{D}}) \in [\![\tau]\!]_{\mathcal{I}_{\mathsf{ty}},[\vec{\alpha}\mapsto\vec{\mathcal{D}}]}$, We require that the logical symbols are interpreted in the usual way in terms of 0 and 1. For example, $\mathcal{J}(\bot) = 0$ and $\mathcal{J}(\mathsf{V})(v, w) = \max\{v, w\}$. Note that in the presence of $\boldsymbol{\varepsilon}$ in the signature, every type τ must be interpreted by a nonempty set, for $\mathcal{J}(\boldsymbol{\varepsilon}\langle\tau\rangle\ (\lambda x.\, \bot))$ to be defined.

The *comprehension principle* states that every function designated by a λ-abstraction is contained in the domain associated with its type. We initially allow λ-abstractions to designate arbitrary elements of the domain. This enables us to define the denotation of a term. Then we impose restrictions to rule out undesirable λ-abstraction designations.

A *λ-designation function* $\mathcal{L}$ for a type interpretation $\mathcal{I}_{\mathsf{ty}}$ is a function that maps a valuation ξ and a λ-abstraction of type τ to an element of $[\![\tau]\!]_{\mathcal{I}_{\mathsf{ty}},\xi}$. An *interpretation* $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}, \mathcal{L})$ combines a type interpretation, an interpretation function, and a λ-designation function.

For an interpretation $\mathcal{I}$ and a valuation ξ, the denotation of a term is defined recursively as $[\![x]\!]_{\mathcal{I},\xi} = \xi(x)$, $[\![\mathsf{f}\langle\vec{\tau}\rangle]\!]_{\mathcal{I},\xi} = \mathcal{J}(\mathsf{f}, [\![\vec{\tau}]\!]_{\mathcal{I}_{\mathsf{ty}},\xi})$, $[\![s\ t]\!]_{\mathcal{I},\xi} = [\![s]\!]_{\mathcal{I},\xi}([\![t]\!]_{\mathcal{I},\xi})$, and $[\![\lambda x.\, t]\!]_{\mathcal{I},\xi} = \mathcal{L}(\xi, \lambda x.\, t)$. For monomorphic closed terms t, the denotation does not depend on the valuation ξ, allowing us to write $[\![t]\!]_{\mathcal{I}}$ instead of $[\![t]\!]_{\mathcal{I},\xi}$.

An interpretation $\mathcal{I}$ is *proper* if $[\![\lambda x : \upsilon.\, t]\!]_{\mathcal{I},\xi}(v) = [\![t]\!]_{\mathcal{I},\xi[x\mapsto v]}$ for every λ-abstraction $\lambda x : \upsilon.\, t$, every valuation ξ, and every value $v \in [\![\upsilon]\!]_{\mathcal{I}_{\mathsf{ty}},\xi}$. We will assume throughout that all interpretations are proper and will construct only proper interpretations. If a type interpretation $\mathcal{I}_{\mathsf{ty}}$ and an interpretation function $\mathcal{J}$ can be extended by a λ-designation function $\mathcal{L}$ to an interpretation $(\mathcal{I}_{\mathsf{ty}}, \mathcal{J}, \mathcal{L})$, then this $\mathcal{L}$ is unique [Fit02, Proposition 2.18].

Given an interpretation $\mathcal{I}$ and a valuation ξ, an equation $s \approx t$ is true if $[\![s]\!]_{\mathcal{I},\xi}$ and $[\![t]\!]_{\mathcal{I},\xi}$ are equal and it is false otherwise. A disequation $s \not\approx t$ is true if $s \approx t$ is false. A clause is true if at least one of its literals is true. A clause set is true if all the clauses it contains are true. An interpretation $\mathcal{I}$ is a *model* of a clause set N, written $\mathcal{I} \models N$, if N is true in $\mathcal{I}$ for every valuation ξ.

A clause C is a *tautology* if $\mathcal{I} \models \{C\}$ for every interpretation $\mathcal{I}$. It is *satisfiable* if there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \{C\}$; otherwise, it is *unsatisfiable*. Notice that these concepts are defined with respect to general, and not necessarily standard, interpretations.

It is sometimes convenient to assume that the interpretation of monomorphic types is injective on types—that is, for all monomorphic τ, υ, if $[\![\tau]\!]_{\mathcal{I}_{\mathsf{ty}}} = [\![\upsilon]\!]_{\mathcal{I}_{\mathsf{ty}}}$, then $\tau = \upsilon$. This assumption is reasonable because the elements of $\mathcal{J}_{\mathsf{ty}}(\kappa)$, where $\kappa \notin \{o, \to\}$, are immaterial and can be renamed if desired and because the set-theoretic representation of functions, as

nonempty sets of pairs, preserves this property. (The sets of pairs are nonempty thanks to the presence of $\boldsymbol{\varepsilon}$ in the signature, as noted above.) We call this principle the *distinct domain assumption*.

3. Hidden-Literal-Based Elimination

In propositional [HJB11] and clausal first-order logic [VBH21a], a hidden literal for a literal L and a clause set N is a literal that can be added or removed from any clause containing L without affecting its truth value in models of N. Several elimination techniques are based on hidden literal; in particular, hidden literal elimination removes hidden literals from clauses in which they occur.

Example 3.1. Consider the literal c and the clause set $N = \{\neg\, \mathsf{a} \vee \mathsf{b},\ \neg\, \mathsf{b} \vee \mathsf{c}\}$. Then b is a hidden literal: Since b implies c according to N, we have that $\mathsf{b} \vee \mathsf{c}$ and c have the same truth value in models of N. Similarly, since a implies c (by transitivity), a is also a hidden literal. Thus, hidden literal elimination would reduce the clause $\mathsf{a} \vee \mathsf{b} \vee \mathsf{c}$ to c.

The first-order definitions of hidden literals, hidden tautologies, hidden literal elimination, hidden tautology elimination, failed literal elimination, hidden tautology reduction, and failed literal reduction [VBH21a] work verbatim in clausal higher-order logic, for both preprocessing and inprocessing. All the techniques preserve satisfiability and unsatisfiability. We call them collectively *hidden-literal-based elimination* (HLBE).

4. Predicate Elimination

Predicate elimination (PE) [GO92, KK16, VBH21a] is a set of techniques that remove all occurrences of some predicate symbol in a first-order problem by resolving clauses that contain it. Predicate elimination generalizes variable elimination in propositional logic [SP04, CS00]. In this section, we generalize two specific techniques to higher-order logic.

4.1. Singular Predicate Elimination.

The first technique is called singular (or "non-self-referential") predicate elimination. The definitions below are adapted from monomorphic first-order logic.

Definition 4.1. A predicate symbol p *occurs deep* in a clause C if it occurs in a position other than as the head of the atom of a p-literal somewhere in C. The symbol p *occurs deep* in a clause set N if it occurs deep in one of its clauses $C \in N$.

Definition 4.2. A predicate symbol p is called *singular* for a clause C if these conditions are met:

(1) C contains at most one p-literal;
(2) p does not occur deep in C.

The symbol p is *singular* for a clause set N if p is singular for every clause contained in N.

Definition 4.3. The clause $C = (\neg)\, \mathsf{p}\langle\vec{\tau}\rangle\ \vec{t} \vee C'$ is *polymorphism-safe* for its literal $(\neg)\, \mathsf{p}\langle\vec{\tau}\rangle\ \vec{t}$ if all type variables occurring in C occur in $\vec{\tau}$. A clause C is *polymorphism-safe* for p if it is polymorphism-safe for all its p-literals. A clause set N is *polymorphism-safe* for p if all the clauses it contains are polymorphism-safe for p.

Definition 4.4. Let $C = \mathsf{p}\langle\vec{\tau}\rangle\ \vec{s}_n \vee C'$ and $D = \neg\,\mathsf{p}\langle\vec{v}\rangle\ \vec{t}_n \vee D'$. The *flat resolvent* of C and D on $\mathsf{p}\langle\vec{\tau}\rangle\ \vec{s}_n$ and $\neg\,\mathsf{p}\langle\vec{v}\rangle\ \vec{t}_n$ is the clause $(s_1 \not\approx t_1 \vee \cdots \vee s_n \not\approx t_n \vee C' \vee D')\sigma$, where σ is the most general unifier of $\vec{\tau} \stackrel{?}{=} \vec{v}$. The flat resolvent is not defined if $\vec{\tau}$ and $\vec{v}$ are not unifiable.

Definition 4.5. 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) $(M, \{(\neg)\,\mathsf{p}\langle\vec{\tau}\rangle\ \vec{s} \vee C'\} \uplus N) \rightsquigarrow (M, N' \cup N)$ if N' is the set that consists of all clauses, up to variable renaming, that are flat resolvents on $(\neg)\,\mathsf{p}\langle\vec{\tau}\rangle\ \vec{s}$ and $(\neg)\,\neg\,\mathsf{p}\langle\vec{v}\rangle\ \vec{t}$ of $(\neg)\,\mathsf{p}\langle\vec{\tau}\rangle\ \vec{s} \vee C'$ and a clause $(\neg)\,\neg\,\mathsf{p}\langle\vec{v}\rangle\ \vec{t} \vee D'$ from M as premises. The premises' variables are renamed apart.

(2) $(M, N) \rightsquigarrow N$ if N contains no p-literals.

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

For finite sets M, N, the resolved set N' is reached in a finite number of steps, and it is unique up to variable renaming. The argument is as for first-order logic [VBH21b, Lemma 13]. Note that the result may contain deep occurrences of p if the initial set N contains such occurrences.

Definition 4.6. Let N be a clause set and p be a singular predicate symbol for N. Let N^+_{p} consist of all clauses belonging to N that contain a positive p-literal, let N^-_{p} consist of all clauses belonging to N that contain a negative p-literal, let $N_{\mathsf{p}} = N^+_{\mathsf{p}} \cup N^-_{\mathsf{p}}$, and let $\overline{N}_{\mathsf{p}} = N \setminus N_{\mathsf{p}}$.

Definition 4.7. Let N be a finite clause set that is polymorphism-safe for p and p be a singular predicate for N. *Singular predicate elimination* (SPE) of p in N replaces N by the $(\Sigma \setminus \{\mathsf{p}\})$-clause set $\overline{N}_{\mathsf{p}} \cup (N^+_{\mathsf{p}} \rtimes_{\mathsf{p}} N^-_{\mathsf{p}})$.

SPE preserves satisfiability: The only clauses added are flat resolvents, and flat resolution is clearly sound. In first-order logic, SPE also preserves unsatisfiability [KK16, Theorem 1]. With a small restriction on polymorphism, this result extends to polymorphic higher-order logic. Also note that deep occurrences of p are not possible in the result, because of the requirement that p be a singular predicate for the input N.

Example 4.8. Thanks to the use of flat resolvents, the unification work is left to the proof calculus. This is convenient, because higher-order unification is undecidable and hence, in general, could not be done exhaustively in a preprocessing technique.

Consider the clause set $N = \{\mathsf{p}\ z\ z \vee \mathsf{q}\ z, \neg\,\mathsf{p}\ (\mathsf{f}\ (y\ \mathsf{a}))\ (y\ (\mathsf{f}\ \mathsf{a})), \neg\,\mathsf{q}\ \mathsf{b}\}$. Applying SPE to p transforms N into $N' = \{\mathsf{f}\ (y\ \mathsf{a}) \not\approx z \vee y\ (\mathsf{f}\ \mathsf{a}) \not\approx z \vee \mathsf{q}\ z, \neg\,\mathsf{q}\ \mathsf{b}\}$. It is then the calculus's task to enumerate values for z that solve the unification problem $z \stackrel{?}{=} \mathsf{f}\ (y\ \mathsf{a}) \stackrel{?}{=} y\ (\mathsf{f}\ \mathsf{a})$. These values are

$$\mathsf{f}\ \mathsf{a},\ \mathsf{f}\ (\mathsf{f}\ \mathsf{a}),\ \mathsf{f}\ (\mathsf{f}\ (\mathsf{f}\ \mathsf{a})),\ \ldots$$

In λ-superposition [BBTV21], this enumeration would be the responsibility of the ERes inference rule. The first clause in N' could be simplified to $y\ (\mathsf{f}\ \mathsf{a}) \not\approx \mathsf{f}\ (y\ \mathsf{a}) \vee \mathsf{q}\ (\mathsf{f}\ (y\ \mathsf{a}))$, by eliminating z. Then ERes would unify the two sides of the first literal, producing one conclusion per unifier. Since there are infinitely many unifiers, this would lead to infinitely many conclusions:

$$\mathsf{q}\ (\mathsf{f}\ \mathsf{a}),\ \mathsf{q}\ (\mathsf{f}\ (\mathsf{f}\ \mathsf{a})),\ \mathsf{q}\ (\mathsf{f}\ (\mathsf{f}\ (\mathsf{f}\ \mathsf{a}))),\ \ldots$$

Using dovetailing, this infinite enumeration can be interwoven with other inferences and other activities of the prover [VBB+21, Section 5].

Example 4.9. Consider the satisfiable clause set $N = \{\mathsf{p}\ (\mathsf{f}\ z) \vee \mathsf{q}\ z, \neg\ \mathsf{p}\ (\mathsf{f}\ \mathsf{a})\}$. SPE of p transforms N into the equally satisfiable set $N' = \{\mathsf{f}\ \mathsf{a} \not\approx \mathsf{f}\ z \vee \mathsf{q}\ z\}$.

Note that although p is absent from N', a predicate that meets its specification can be created based on the first clause of N, in which the p-literal is positive. This predicate is $\lambda x.\ \exists y.\ x \approx \mathsf{f}\ y \wedge \neg\ \mathsf{q}\ y$. In general, we would use all the clauses in which the p-literal is positive and ignore the other clauses. (Dually, we could have defined the predicate in terms of the clauses in which the p-literal is negative.) This predicate makes p true only when it must be true to satisfy the first clause—namely, when the clause's non-p-literal is false.

If we replace p with this λ-abstraction in N and β-reduce, we obtain the satisfiable set $N'' = \{(\exists y.\ \mathsf{f}\ z \approx \mathsf{f}\ y \wedge \neg\ \mathsf{q}\ y) \vee \mathsf{q}\ z, \neg\ (\exists y.\ \mathsf{f}\ \mathsf{a} \approx \mathsf{f}\ y \wedge \neg\ \mathsf{q}\ y)\}$. Nothing essential is lost by eliminating p—if we need p, we can use the λ-abstraction. This idea is the key to the proof of Theorem 4.11 below.

Example 4.10. Consider the clause set $N = \{\neg\ y\ \mathsf{a}, \mathsf{p}\ \mathsf{a}\}$. It is easy to see that the set is unsatisfiable, by taking $y := \mathsf{p}$. SPE of p transforms N into the set $N' = \{\neg\ y\ \mathsf{a}\}$. Although $y\ \mathsf{a}$ is unifiable with the literal $\mathsf{p}\ \mathsf{a}$, the first clause is left unchanged by SPE. Like N, N' is unsatisfiable. We cannot witness unsatisfiability by taking $y := \mathsf{p}$. We could try to take $y := \lambda x.\ x \approx \mathsf{a}$, simulating p on the input a, but in the absence of $\mathsf{p}\ \mathsf{a}$ in N' this would not suffice. Instead, we take $y := \lambda x.\ \mathsf{T}$. Indeed, we could have taken this instantiation for y to show that N is unsatisfiable, without exploiting the presence of $\mathsf{p}\ \mathsf{a}$.

Example 4.10 corroborates our choice of ignoring literals headed by a variable in the definition of SPE, focusing instead on p-literals. The intuition is that often p is unnecessary to have unsatisfiability, and when it is necessary it can be simulated by a λ-abstraction that does not contain it.

Theorem 4.11. *Let N be a finite clause set that is polymorphism-safe for p and p be a singular predicate symbol for N. Let N' be the result of applying SPE of p to N. Then N' is satisfiable if and only if N is satisfiable.*

Proof. The "if" direction follows immediately from the soundness of flat resolution. For the other direction, our strategy is inspired by Khasidashvili and Korovin [KK16, Theorem 1].

Let $\mathsf{p} : \Pi\vec{\alpha}_m.\tau_1 \to \cdots \to \tau_n \to o$. Let $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}, \mathcal{L})$ be a model of $N' = \overline{N}_{\mathsf{p}} \cup (N^+_{\mathsf{p}} \rtimes_{\mathsf{p}} N^-_{\mathsf{p}})$, with $\mathcal{I}_{\mathsf{ty}} = (\mathcal{U}, \mathcal{J}_{\mathsf{ty}})$. We assume without loss of generality that this model satisfies the distinct domain assumption (Section 2). We will define a new interpretation $\mathcal{I}'$ that extends $\mathcal{I}$ with a semantics for p and show that $\mathcal{I}'$ is a model of $N = \overline{N}_{\mathsf{p}} \cup N^+_{\mathsf{p}} \cup N^-_{\mathsf{p}}$. To achieve this, we will separately show that (1) $\mathcal{I}' \models \overline{N}_{\mathsf{p}}$; (2) $\mathcal{I}' \models N^+_{\mathsf{p}}$; and (3) $\mathcal{I}' \models N^-_{\mathsf{p}}$.

The interpretation $\mathcal{I}' = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}', \mathcal{L}')$ we construct is identical to $\mathcal{I}$ except that $\mathcal{J}'$ is extended so that $\mathcal{J}'(\mathsf{p}, \vec{\mathcal{D}})$ and $\mathcal{L}'$ are defined as follows, by mutual recursion.

We start with $\mathcal{J}'(\mathsf{p}, \vec{\mathcal{D}})$. We construct a suitable interpretation for p as a curried n-ary predicate. Let $\vec{\tau'}$ be monomorphic types such that $[\![\vec{\tau'}]\!]_{\mathcal{I}_{\mathsf{ty}}} = \vec{\mathcal{D}}$. By the distinct domain assumption, these types are unique if they exists. If no such types exists, let $\mathcal{J}'(\mathsf{p}, \vec{\mathcal{D}})$ be the predicate $[\![\lambda\vec{x}.\ \bot]\!]_{\mathcal{I}}$. This predicate is guaranteed to exist in the interpretation of the type of $\mathsf{p}\langle\vec{\tau'}\rangle$ thanks to the comprehension principle (Section 2). The predicate plays the role of a placeholder for impossible interpretations of p; its value is irrelevant.

In the case where the types $\vec{\tau'}$ exist, we will construct a right-hand side for $\mathcal{J}'(\mathsf{p}, \vec{\mathcal{D}})$ using the same idea as in Example 4.9. To cope with polymorphism, we will filter out any

clauses whose p's type arguments cannot be instantiated to $\vec{\tau'}$ and instantiate the remaining clauses.

More precisely, let M be the smallest set such that for each clause $C = \mathsf{p}\langle\vec{v}\rangle\ \vec{t}_n \vee C'$ contained in N_{p}, if there exists a substitution σ such that $\vec{v}\sigma = \vec{\tau'}$, then have M contain $C\sigma$. Notice that the polymorphism-safety hypothesis ensures that $C\sigma$ is monomorphic and uniquely specified. This is desirable because we want to assign a unique right-hand side to $\mathcal{J}'(\mathsf{p}, \vec{\mathcal{D}})$.

We now define a term u whose interpretation $[\![u]\!]_{\mathcal{I}} = [\![u]\!]_{\mathcal{I}'}$ will give us the right-hand side. Let $\vec{x}_n$ be a tuple of fresh variables. With each clause $\mathsf{p}\langle\vec{\tau'}\rangle\ \vec{t}_n \vee C'$ contained in M and whose free variables are $\vec{y}$, associate the formula

$$\mathbf{\exists}\vec{y}.\, x_1 \mathbin{\boldsymbol{\approx}} t_1 \mathbin{\mathbf{\wedge}} \cdots \mathbin{\mathbf{\wedge}} x_n \mathbin{\boldsymbol{\approx}} t_n \mathbin{\mathbf{\wedge}} \mathbf{\neg}\, [C']$$

(Recall from Section 2 that $[C']$ denotes a formula representing the clause C'.) Let $\varphi_1, \ldots, \varphi_k$ be all such formulas, and let $\varphi = \varphi_1 \mathbin{\mathbf{\vee}} \cdots \mathbin{\mathbf{\vee}} \varphi_k$. Then we take $u = \lambda\vec{x}.\, \varphi$. This choice of u will ensure that $\mathcal{I}'$ satisfies every clause in N_{p}^+, and thanks to the comprehension principle, the predicate denoted by u is guaranteed to exist in the interpretation of $\tau'_1 \to \cdots \to \tau'_n \to o$.

To finish the definition of $\mathcal{I}'$, we must specify $\mathcal{L}'$. Given a term t, let $t[u/\mathsf{p}]$ denote the variant of the term t in which all occurrences of p are replaced by the term u as defined above (for suitable types $\vec{\tau'}$). For all valuations ξ and λ-abstractions $\lambda x : v.\, t$, we define $\mathcal{L}'(\xi, \lambda x.\, t)$ as the function that maps each $v \in [\![v]\!]_{\mathcal{I}_{\mathsf{ty}},\xi}$ to $[\![t[u/\mathsf{p}]]\!]_{\mathcal{I},\xi[x\mapsto v]}$. This function exists in the domain associated with the λ-abstraction's type because $\mathcal{I}$ obeys the comprehension principle. Moreover, because $t[u/\mathsf{p}]$ replaces p by a term with the same semantics according to $\mathcal{I}'$, the interpretation $\mathcal{I}'$ is proper.

We are now ready to tackle the three conditions we need to prove. To prove (1), we start from $\mathcal{I} \models \overline{N}_{\mathsf{p}}$ and show $\mathcal{I}' \models \overline{N}_{\mathsf{p}}$. More precisely, we must show that $[\![\overline{N}_{\mathsf{p}}]\!]_{\mathcal{I}',\xi}$ is true for any valuation ξ. This is obvious because $\mathcal{I}$ and $\mathcal{I}'$ only differ on p, which does not occur in $\overline{N}_{\mathsf{p}}$.

To prove (2), we show that $\mathcal{I}' \models N_{\mathsf{p}}^+$ holds by construction of $\mathcal{I}'$. More precisely, we must show that $[\![N_{\mathsf{p}}^+]\!]_{\mathcal{I}',\xi}$ is true for any valuation ξ. Let $C = \mathsf{p}\langle\vec{v}\rangle\ \vec{t} \vee C'$ be a clause in N_{p}^+. By definition,

$$\mathcal{J}'(\mathsf{p}, [\![\vec{v}]\!]_{\mathcal{I}_{\mathsf{ty}},\xi}) = [\![\lambda\vec{x}.\, \cdots \mathbin{\mathbf{\vee}} \underbrace{(\mathbf{\exists}\vec{y}.\, x_1 \mathbin{\boldsymbol{\approx}} t_1 \mathbin{\mathbf{\wedge}} \cdots \mathbin{\mathbf{\wedge}} x_n \mathbin{\boldsymbol{\approx}} t_n \mathbin{\mathbf{\wedge}} \mathbf{\neg}\, [C'])}_{\psi} \mathbin{\mathbf{\vee}} \cdots]\!]_{\mathcal{I},\xi}$$

If $[\![C']\!]_{\mathcal{I}',\xi}$ is true, then clearly $[\![C]\!]_{\mathcal{I}',\xi}$ is true. Otherwise, the literal $\mathsf{p}\langle\vec{v}\rangle\ \vec{t}$ is true, because the disjunct ψ is true. This can be seen by taking the values of $\vec{y}$ under ξ as the existential witnesses. The arguments $\vec{t}$ in $\mathsf{p}\langle\vec{v}\rangle\ \vec{t}$ passed for $\vec{x}$ match those expected by the equalities $x_i \mathbin{\boldsymbol{\approx}} t_i$, and since $[\![C']\!]_{\mathcal{I}',\xi}$ is false, $[\![\mathbf{\neg}\, [C']]\!]_{\mathcal{I}',\xi}$ is true.

For the proof of (3), the argument is similar to Khasidashvili and Korovin's [KK16, Theorem 1]. It relies on the presence of the flat resolvents $N_{\mathsf{p}}^+ \rtimes_{\mathsf{p}} N_{\mathsf{p}}^-$ in the result set. Let $D = \neg\, \mathsf{p}\langle\vec{v}\rangle\ \vec{u} \vee D'$ be a clause in N_{p}^-. If $[\![D']\!]_{\mathcal{I},\xi}$ is true, then clearly $[\![D']\!]_{\mathcal{I}',\xi}$ is true. Otherwise, $[\![D']\!]_{\mathcal{I}',\xi}$ is false, and $[\![\neg\, \mathsf{p}\langle\vec{v}\rangle\ \vec{u}]\!]_{\mathcal{I}',\xi}$ is either true or false. In the true case, $[\![D]\!]_{\mathcal{I}',\xi}$ is true, as desired. As for the remaining case, it is impossible for the following reason. If $[\![\neg\, \mathsf{p}\langle\vec{v}\rangle\ \vec{u}]\!]_{\mathcal{I}',\xi}$ were false, this would mean $[\![\mathsf{p}\langle\vec{v}\rangle\ \vec{u}]\!]_{\mathcal{I}',\xi}$ is true. By definition of $\mathcal{I}'$, this would then mean $[\![\cdots \mathbin{\mathbf{\vee}} (\mathbf{\exists}\vec{y}.\, u_1 \mathbin{\boldsymbol{\approx}} t_1 \mathbin{\mathbf{\wedge}} \cdots \mathbin{\mathbf{\wedge}} u_n \mathbin{\boldsymbol{\approx}} t_n \mathbin{\mathbf{\wedge}} \mathbf{\neg}\, [C']) \mathbin{\mathbf{\vee}} \cdots]\!]_{\mathcal{I},\xi}$ is true.

Suppose the displayed disjunct is one of those that makes the whole big disjunction true. This means that there exists a clause $C = \mathsf{p}\langle\vec{\tau}\rangle\ \vec{t} \vee C'$ in N_{p}^+ such that $[\![\vec{t}]\!]_{\mathcal{I}',\xi} = [\![\vec{u}]\!]_{\mathcal{I}',\xi}$,

and $[\![C']\!]_{\mathcal{I}',\xi}$ is false, where $\vec{\tau}$ is unifiable with $\vec{v}$. Let σ be the most general unifier of $\vec{\tau} \stackrel{?}{=} \vec{v}$. If C exists, the flat resolvent $(t_1 \not\approx u_1 \vee \cdots \vee t_n \not\approx u_n \vee C' \vee D')\sigma$ of C and D must be false in $\mathcal{I}'$ under ξ, and since it does not contain p, it would be false in $\mathcal{I}$ under ξ as well, contradicting the hypothesis that $\mathcal{I}$ satisfies it. □

As Khasidashvili and Korovin observed, eliminating all singular predicates indiscriminately can dramatically increase the number of clauses in the problem. To prevent this explosion, Vukmirović et al. proposed the following criterion. Let $K_{\text{tol}} \in \mathbb{N}$ be a tolerance parameter. The application of SPE from N to N' is allowed if $\lambda(N') < \lambda(N) + K_{\text{tol}}$ or $\mu(N') < \mu(N)$ or $|N'| < |N| + K_{\text{tol}}$, 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 unique variables in C.

4.2. **Defined Predicate Elimination.** The next technique we generalize from first-order logic to higher-order logic is called defined predicate elimination. It generalizes, in turn, the propositional technique of elimination by substitution [EB05].

Given a clause set N, the basic idea is that the set N_{p} of clauses containing p is partitioned between a definition set (or "gate") G and the remaining clauses R. The definition set fully characterizes p for all input in a unique way and can be seen as constituting a definition of the form $\mathsf{p}\ \vec{x} \leftrightarrow \varphi$, where the variables $\vec{x}$ are distinct, p does not occur in φ, and the variables in φ are all among $\vec{x}$. Because of clausification, G will usually consist of multiple clauses that together are equivalent to $\mathsf{p}\ \vec{x} \leftrightarrow \varphi$ for some φ.

We define definition sets largely as in monomorphic first-order logic, but with additional requirements on the type arguments and type variables.

Definition 4.12. Let G be a clause set and p be a predicate symbol. The set G is a *definition set* for p if

(1) p is singular for G;
(2) G consists of clauses of the form $(\neg)\ \mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \vee C'$ up to variable renaming, where $\vec{\alpha}$ are distinct type variables and $\vec{x}$ are distinct term variables;
(3) the type variables in C' are all among $\vec{\alpha}$;
(4) the term variables in C' are all among $\vec{x}$;
(5) all clauses in $G^+_{\mathsf{p}} \rtimes_{\mathsf{p}} G^-_{\mathsf{p}}$ are tautologies; and
(6) $E(\vec{\tau}, \vec{\mathsf{c}})$ is unsatisfiable, where the *environment* $E(\vec{\alpha}, \vec{x})$ consists of all subclauses C' of any $(\neg)\ \mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \vee C' \in G$, $\vec{\tau}$ is a tuple of distinct nullary type constructors substituted in for $\vec{\alpha}$, and $\vec{\mathsf{c}}$ is a tuple of distinct fresh symbols substituted in for $\vec{x}$.

Intuitively, conditions 1 and 2 check that the definition set looks like the clausification of a definition; conditions 3, 4, and 5 check that the definition is not overconstrained (e.g., $\mathsf{p}\,\mathsf{a}$ is not required to be both true and false); and condition 6 checks that it is not underconstrained (e.g., $\mathsf{p}\,\mathsf{a}$ is not unspecified).

Definition 4.13. Given a definition set G for p, its *associated definition* is the formula $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \leftrightarrow \varphi$, up to variable renaming, where φ is the disjunction $\varphi_1 \vee \cdots \vee \varphi_n$ of all formulas φ_j of the form $\neg\,[C']$ such that $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \vee C'$ is contained in G up to variable renaming.

Note that by the definition of definition set, in Definition 4.13 the type variables $\vec{\alpha}$ are distinct, the term variables $\vec{x}$ are distinct, p does not occur in φ, the type variables in φ are all among $\vec{\alpha}$, and the variables in φ are all among $\vec{x}$.

Example 4.14. For the formula $\mathsf{p}\,x\,y \leftrightarrow \mathsf{q}\,x \vee \mathsf{r}\,y$, clausification would produce the clause set $\{\neg\,\mathsf{p}\,x\,y \vee \mathsf{q}\,x \vee \mathsf{r}\,y,\ \mathsf{p}\,x\,y \vee \neg\,\mathsf{q}\,x,\ \mathsf{p}\,x\,y \vee \neg\,\mathsf{r}\,y\}$, which qualifies as a definition set for p. The associated definition is $\mathsf{p}\,x\,y \leftrightarrow \neg\neg\,\mathsf{q}\,x \mathbin{\mathsf{V}} \neg\neg\,\mathsf{r}\,y$.

Lemma 4.15. *Let G be a definition set for p. Then G is equivalent to the definition $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \leftrightarrow \varphi$ associated with G.*

Proof. We will show that under any valuation ξ, any model of G is a model of $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \leftrightarrow \varphi$ and vice versa.

Let $\mathcal{I} \models G$. We will show that $[\![\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x}]\!]_{\mathcal{I},\xi} = [\![\varphi]\!]_{\mathcal{I},\xi}$. If $[\![\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x}]\!]_{\mathcal{I},\xi}$ is false, then for each clause $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \vee C' \in G$, we have that $[\![C']\!]_{\mathcal{I},\xi}$ must be true. This in turn makes the right-hand side φ false. Otherwise, $[\![\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x}]\!]_{\mathcal{I},\xi}$ is true. Then for each clause $\neg\,\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \vee C' \in G$, we have that $[\![C']\!]_{\mathcal{I},\xi}$ must be true. By condition 6, there must exist a clause $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \vee D' \in G$ such that $[\![D']\!]_{\mathcal{I},\xi}$ is false. This means that $[\![\neg\,D']\!]_{\mathcal{I},\xi}$ is true and hence the entire disjunction φ is true, as desired.

Now let $\mathcal{I} \models \mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \leftrightarrow \varphi$. We need to show that $\mathcal{I} \models G$. We will first prove the case of clauses in G where the p-literal is positive; then we will consider the negative case. Let $C = \mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \vee C' \in G$. If $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x}$ is true, then C is true, as desired. Otherwise, from $\mathcal{I} \models \mathsf{p}\langle\vec{\alpha}\rangle\,\vec{x} \leftrightarrow \varphi$ we have that φ is false. This means that all of its disjuncts are false and hence that C' is true, meaning that C is true. For the remaining case, let $C = \neg\,\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \vee C' \in G$. If $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x}$ is false, then C is true, as desired. Otherwise, from $\mathcal{I} \models \mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \leftrightarrow \varphi$ we have that φ is true. This means that there exists a clause $D = \mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \vee D' \in G$ such that $[\![D']\!]_{\mathcal{I},\xi}$ is false. By condition 5, the resolvent $C' \vee D'$ of C and D must be a tautology. Hence C' is true and thus C is true, as desired. □

Once a definition is identified, it is expanded in the remaining clauses R. For p-literals in R, this is achieved as in first-order logic using flat resolution. For deeper occurrences of $\mathsf{p}\langle\vec{\tau}\rangle$ in R, which may arise in higher-order logic, this is achieved by replacing them by the λ-abstraction $\lambda\vec{x}.\,\varphi\{\vec{\alpha} \mapsto \vec{\tau}\}$. An alternative would be to replace *all* occurrences of $\mathsf{p}\langle\vec{\tau}\rangle$ and not only deep occurrences, but this would leave more work for the clausifier.

Definition 4.16. Let N be a clause set and p be a predicate symbol. Let $G \subseteq N$ be a definition set for p with associated definition $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \leftrightarrow \varphi$. Let $R = N_{\mathsf{p}} \setminus G$. *Defined predicate elimination* (DPE) of p in N replaces N by $\overline{N}_{\mathsf{p}} \cup (G \rtimes_{\mathsf{p}} R)[\mathsf{p}\langle\vec{\tau}\rangle \mapsto \lambda\vec{x}.\,\varphi\{\vec{\alpha} \mapsto \vec{\tau}\}]$.

The key result is that DPE preserves satisfiability and unsatisfiability. The proof builds on three lemmas.

Lemma 4.17. *Let G be a definition set for p and R be an arbitrary clause set. If $(G, R) \rightsquigarrow (G, R')$, then $G \cup R$ and $G \cup R'$ are equivalent.*

Proof. The proof is essentially as in the first-order case [VBH21b, Lemma 20]. □

Lemma 4.18. *Let G be a definition set for p with associated definition $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \leftrightarrow \varphi$, and let R be a clause set. Then $G \cup R$ and $G \cup R[\mathsf{p}\langle\vec{\tau}\rangle \mapsto \lambda\vec{x}.\,\varphi\{\vec{\alpha} \mapsto \vec{\tau}\}]$ are equivalent.*

Proof. By Lemma 4.15, G entails the characterization $\mathsf{p}\langle\vec{\alpha}\rangle\ \vec{x} \leftrightarrow \varphi$. Hence, by functional extensionality, G entails $\mathsf{p}\langle\vec{\alpha}\rangle \approx \lambda\vec{x}.\,\varphi$. Thus, in any model of G, $\mathsf{p}\langle\vec{\alpha}\rangle$ has the same interpretation as $\lambda\vec{x}.\,\varphi$. In particular, this applies to their instances: $\mathsf{p}\langle\vec{\tau}\rangle$ has the same interpretation as $\lambda\vec{x}.\,\varphi\{\vec{\alpha} \mapsto \vec{\tau}\}$. □

Lemma 4.19. *Let G be a definition set for p and R be a clause set with no occurrences of p. Then $G \cup R$ is satisfiable if and only if R is satisfiable.*

Proof. The proof is essentially as in the first-order case [VBH21b, Lemma 21]. □

Theorem 4.20. *The result of applying DPE to a finite clause set N is satisfiable if and only if N is satisfiable.*

Proof. Let p be a predicate symbol and $G \subseteq N$ be the definition set used by DPE. Let $R = N_{\mathsf{p}} \setminus G$. The core of DPE is the computation of $G \rtimes_{\mathsf{p}} R$, via a derivation $(G, R) \rightsquigarrow^n (G, R') \rightsquigarrow R'$. Applying Lemma 4.17 n times, we get that $G \cup R$ is equivalent to $G \cup R'$. Moreover, by Lemma 4.18, $G \cup R'$ is equivalent to $G \cup R'[\mathsf{p}\langle\vec{\tau}\rangle \mapsto \lambda\vec{x}.\, \varphi\{\vec{\alpha} \mapsto \vec{\tau}\}]$. Finally, by Lemma 4.19, $G \cup R'[\mathsf{p}\langle\vec{\tau}\rangle \mapsto \lambda\vec{x}.\, \varphi\{\vec{\alpha} \mapsto \vec{\tau}\}]$ is equivalent to $R'[\mathsf{p}\langle\vec{\tau}\rangle \mapsto \lambda\vec{x}.\, \varphi\{\vec{\alpha} \mapsto \vec{\tau}\}]$. □

4.3. **Portfolio Predicate Elimination.** A reasonable strategy for applying predicate elimination is to use a portfolio of DPE and SPE, first trying to apply DPE and, if this fails, trying SPE as a fallback.

Definition 4.21. 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 applies DPE on p. Otherwise, if p is singular in N, PPE applies SPE on p. In all other cases, PPE is not applicable.

Like SPE and DPE (Theorems 4.11 and 4.20), PPE can be used as a preprocessor without affecting satisfiability. As for inprocessing, Vukmirović et al. [VBH21b] explained that under a reasonable condition, the first-order version of PPE can be used at any point during proof search in a superposition prover without compromising refutational completeness. Inspection of the proofs reveals that the same applies to higher-order PPE and λ-superposition.

5. Blocked Clause Elimination

In propositional logic, a powerful technique for simplifying a clause set is to identify and remove so-called blocked clauses. These are clauses whose resolvents with other clauses in the set are all tautologies. Removing such clauses preserves unsatisfiability. Blocked clause elimination has been extended to first-order logic with equality by Kiesl et al. [KSS+17]. They call their key notion "equality-blocked clauses," but since we consider only a logic with equality, we simply call these clauses "blocked."

Blocked clause elimination has been shown by Vukmirović et al. [VBH21a, Section V] to be incompatible with the saturation loop of a superposition prover. Nevertheless, the technique can still be used effectively as a preprocessor, or even as an inprocessing technique within the prover's saturation loop at the cost of potential divergence on some unsatisfiable problems.

Our extension to polymorphic higher-order logic is based on a slightly weaker definition of blocked clause than Kiesl et al. We were unsuccessful at showing that a generalization of blocked clause elimination based on their concept preserves unsatisfiability with respect to general interpretations. The notion we propose allows the generalization.

Definition 5.1. Let $C = L \vee C'$ and $D = L' \vee D'$ be clauses such that
(1) the atom of L is $\mathsf{p}\langle\vec{\tau}\rangle\ \vec{s}_n$;
(2) the atom of L' is $\mathsf{p}\langle\vec{\upsilon}\rangle\ \vec{t}_n$;
(3) the literal L' is of opposite polarity to L;
(4) C and D have no (type or term) variables in common; and

(5) σ be the most general unifier of $\overrightarrow{\tau} \stackrel{?}{=} \overrightarrow{v}$.

The clause $\big(\big(\bigvee_{j=1}^{n} s_j \not\approx t_j\big) \vee C' \vee D'\big)\sigma$ is a *binary flat L-resolvent* of C and D.

We already see a first key difference with Keisl et al.: They consider n-ary flat resolvents, whereas we need to consider only binary resolvents, for reasons explained below (Example 5.4). Another, more superficial difference is that our definition is generalized to polymorphic higher-order logic.

Definition 5.2. Let $L = (\neg)\,\mathsf{p}\langle\overrightarrow{\tau}\rangle\ \overrightarrow{s}$ be a predicate literal, $C = L \vee C'$ be a clause, and N be a clause set. Let N' consist of all clauses from $N \setminus \{C\}$ with their type and term variables renamed so that N' shares no variables with C. The clause C is *blocked* by L in the set N if the following conditions are met:

(1) C is polymorphism-safe for L;
(2) N contains no deep occurrences of p;
(3) C' contains no p-literals with the same polarity as L; and
(4) all binary flat L-resolvents between C and clauses in N' are tautologies.

We now see another key difference with Keisl et al.: They have no restriction corresponding to condition 3 of Definition 5.2. In this respect, our notion is less powerful than theirs. (They also have no restriction corresponding to conditions 1 and 2, but these conditions are trivially satisfied in a monomorphic first-order setting.)

Example 5.3. This example is based on Keisl et al. [KSS+17, Example 1]. Let $C = \neg\,\mathsf{p} \vee \mathsf{q}$, and take $N = \{C, \mathsf{p} \vee \neg\,\mathsf{q}, \neg\,\mathsf{q} \vee \mathsf{r}\}$ as the clause set. The clause C is blocked by $\neg\,\mathsf{p}$ in N according to Definition 5.2 because the only resolvent of C on $\neg\,\mathsf{p}$ is the tautology $\mathsf{q} \vee \neg\,\mathsf{q}$ resulting from resolution against $\mathsf{p} \vee \neg\,\mathsf{q}$. The clause C is also blocked according to the definition in Kiesl et al.

Example 5.4. The next example is also based on Keisl et al. [KSS+17, Example 4]: $C = \mathsf{p}\ x\ y \vee \mathsf{p}\ y\ x$, $D = \neg\,\mathsf{p}\ x\ y \vee \neg\,\mathsf{p}\ y\ x$, and $N = \{C, D\}$. The set N is unsatisfiable, because C entails $\mathsf{p}\ x\ x$ and D entails $\neg\,\mathsf{p}\ x\ x$. On the other hand, D alone is satisfiable. Hence, removing C from N does *not* preserve unsatisfiability, and therefore C should *not* be considered blocked. With Definition 5.2, C correctly cannot be blocked on a p-literal by condition 3, because of the presence of another p-literal in the clause. With Kiesl et al., this condition is missing, but since they consider all n-ary resolvents, the nontautological resolvent $\mathsf{p}\ x\ x$ is computed. In both cases, C is correctly considered not blocked.

Example 5.5. Let $C = \mathsf{p}\,\mathsf{a} \vee \mathsf{p}\,\mathsf{b} \vee \neg\mathsf{q}$, $D = \neg\mathsf{p}\,x \vee \mathsf{q}$, and $N = \{C, D\}$. With Definition 5.2, condition 3 prevents C from being considered blocked. In contrast, the clause is considered blocked by Kiesl et al.

We will now show that removing a blocked clause from a clause set preserves the set's unsatisfiability. Our strategy is inspired by Kiesl et al. [KSS+17, Section 4].

Definition 5.6. Let $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}, \mathcal{L})$ be an interpretation and $L \vee C'$ be a clause that is polymorphism-safe for L and where $L = (\neg)\,\mathsf{p}\langle\overrightarrow{\tau}\rangle\ \overrightarrow{s}_n$. Let $\overrightarrow{y}$ be the tuple of all free variables in $L \vee C'$ and $\overrightarrow{x}_n$ be a tuple of fresh variables. The interpretation $\mathcal{I}^\star = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}^\star, \mathcal{L}^\star)$ obtained by *flipping* the truth value of L in $L \vee C'$ is defined as follows by mutual recursion. We let $\mathcal{J}^\star$ be the function defined as follows:

$$\mathcal{J}^\star(\mathsf{f}, \overrightarrow{\mathcal{D}}) = \begin{cases} [\![\lambda \overrightarrow{x}_n.\ \varphi]\!]_{\mathcal{I},\xi} & \text{if } \mathsf{f} = \mathsf{p} \text{ and } [\![\overrightarrow{\tau}]\!]_{\mathcal{I}_{\mathsf{ty}},\xi} = \overrightarrow{\mathcal{D}} \text{ for some type valuation } \xi \\ \mathcal{J}(\mathsf{f}, \overrightarrow{\mathcal{D}}) & \text{otherwise} \end{cases}$$

where φ is defined by

$$\varphi = \begin{cases} \mathsf{p}\langle\vec{\tau}\rangle\ \vec{x} \mathbin{\mathsf{V}} (\mathsf{\exists}\vec{y}.\ x_1 \approx s_1 \mathbin{\mathsf{\wedge}} \cdots \mathbin{\mathsf{\wedge}} x_n \approx s_n \mathbin{\mathsf{\wedge}} \mathsf{\neg}\,[C']) & \text{if } L \text{ is positive} \\ \mathsf{p}\langle\vec{\tau}\rangle\ \vec{x} \mathbin{\mathsf{\wedge}} (\mathsf{\forall}\vec{y}.\ x_1 \approx s_1 \mathbin{\mathsf{\wedge}} \cdots \mathbin{\mathsf{\wedge}} x_n \approx s_n \mathbin{\mathsf{\rightarrow}} [C']) & \text{if } L \text{ is negative} \end{cases}$$

Moreover, for all valuations ξ and λ-abstractions $\lambda x : \upsilon.\ t$, we let $\mathcal{L}^\star(\xi, \lambda x.\ t)$ be the function that maps each $v \in [\![\upsilon]\!]_{\mathcal{I}_{\mathsf{ty}},\xi}$ to $[\![t]\!]_{\mathcal{I}^\star,\xi[x\mapsto v]}$.

This definition introduces a well-formed interpretation. Because $L \vee C'$ is polymorphism-safe for L, the right-hand side $[\![\lambda\vec{x}_n.\ \varphi]\!]_{\mathcal{I},\xi}$ of $\mathcal{J}^\star(\mathsf{f}, \vec{\mathcal{D}})$ is uniquely defined. Moreover, the comprehension principle guarantees that the corresponding predicate exists in the interpretation of p's type. Similarly, the function that provides the interpretation for a λ-abstraction $\lambda x.\ t$ exists in the domain associated with the λ-abstraction's type. This is because the semantics in $\mathcal{I}^\star$ of any occurrences of p in t corresponds to the semantics in $\mathcal{I}$ of $\lambda\vec{x}_n.\ \varphi$, and $\mathcal{I}$ is a well-formed interpretation.

The intuition behind $\mathcal{I}^\star$ is that whenever the clause $L \vee C'$ is blocked and $\mathcal{I} \models N \setminus \{L \vee C'\}$, we have $\mathcal{I}^\star \models N$. Since adding the blocked clause preserves satisfiability, removing it preserves unsatisfiability.

Example 5.7. We will try to justify the definition of φ above with an example. Consider the clause set $\{\mathsf{p}\ \mathsf{a},\ \neg\ \mathsf{p}\ z \vee z \approx \mathsf{a} \vee z \approx \mathsf{b}\}$ and an interpretation $\mathcal{I}$ that maps p to the uniformly false predicate. Clearly, $\mathcal{I}$ is not a model of the first clause, $\mathsf{p}\,\mathsf{a}$. The interpretation $\mathcal{I}^\star$ obtained by flipping the truth value of the literal $\mathsf{p}\,\mathsf{a}$ in the first clause interprets p in the same way as $\mathcal{I}$ interprets

$$\lambda x.\ \mathsf{p}\ x \mathbin{\mathsf{V}} (x \approx \mathsf{a} \mathbin{\mathsf{\wedge}} \mathsf{\neg}\,\mathsf{\bot})$$

In other words, $\mathcal{I}^\star$ makes p true for arguments interpreted as equal to a and false otherwise. Intuitively, the interpretation $\mathcal{I}$ is "flipped" to satisfy the clause $\mathsf{p}\,\mathsf{a}$. The resulting interpretation $\mathcal{I}^\star$ makes the clause "more true" than $\mathcal{I}$.

Next, assume instead that $\mathcal{I}$ interprets p as the uniformly true predicate, and consider the interpretation $\mathcal{I}^\star$ obtained by flipping the truth value of $\neg\ \mathsf{p}\ z$ in the second clause, $\neg\ \mathsf{p}\ z \vee z \approx \mathsf{a} \vee z \approx \mathsf{b}$. Then $\mathcal{I}^\star$ interprets p in the same way as $\mathcal{I}$ interprets

$$\lambda x.\ \mathsf{p}\ x \mathbin{\mathsf{\wedge}} (\mathsf{\forall} z.\ x \approx z \mathbin{\mathsf{\rightarrow}} z \approx \mathsf{a} \mathbin{\mathsf{V}} z \approx \mathsf{b})$$

This means that $\mathcal{I}^\star$ makes p true for arguments interpreted as equal to a or b and false otherwise. Intuitively, the interpretation $\mathcal{I}$ is "flipped" to satisfy the clause $\neg\mathsf{p}z \vee z{\approx}\mathsf{a} \vee z{\approx}\mathsf{b}$; whenever $z \not\approx \mathsf{a}$ and $z \not\approx \mathsf{b}$, we have $\neg\ \mathsf{p}\ z$. The resulting interpretation $\mathcal{I}^\star$ makes the clause "more true" than $\mathcal{I}$.

We now introduce an auxiliary concept that will help bridge the gap between $\mathcal{I}$ and $\mathcal{I}^\star$. We will find it useful to compare the interpretations of tuples of types or terms using some well-order. The precise order used will not matter. Given a set A, we denote by $\sqsubset_A$ an arbitrary but fixed strict well-order (i.e., well-founded strict total order), and by $\sqsubseteq_A$ the corresponding nonstrict well-order. The order $\sqsubset_A$ is guaranteed to exist by the well-ordering theorem (also called Zermelo's theorem). We will sometimes omit the subscript A when it can be inferred from the context.

Definition 5.8. Let $\mathsf{p} : \Pi\vec{\alpha}_m.\ \tau_1 \to \cdots \to \tau_n \to o$ be a predicate symbol. Let $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}, \mathcal{L})$ be an interpretation, and let $\mathcal{I}^\star = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}^\star, \mathcal{L}^\star)$ be an interpretation obtained by flipping the truth value of a p-literal. Let $\vec{\mathcal{E}} \in \mathcal{U}^m$, $\xi = [\vec{\alpha} \mapsto \vec{\mathcal{E}}]$, $W = [\![\tau_1]\!]_{\mathcal{I}_{\mathsf{ty}},\xi} \times \cdots \times [\![\tau_n]\!]_{\mathcal{I}_{\mathsf{ty}},\xi}$, and $\vec{w} \in W$. The *pseudo-interpretation* $\mathcal{I}^{\vec{\mathcal{E}},\vec{w}} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}^{\vec{\mathcal{E}},\vec{w}}, \mathcal{L}^{\vec{\mathcal{E}},\vec{w}})$ *up to* $\vec{\mathcal{E}}$ and $\vec{w}$ and associated

with N, C, and L is defined as follows by mutual recursion. We let $\mathcal{J}^{\vec{\mathcal{E}},\vec{w}}$ be the function defined as follows:

$$\mathcal{J}^{\vec{\mathcal{E}},\vec{w}}(\mathsf{f},\vec{\mathcal{D}}) = \vec{v}_n \mapsto \begin{cases} \mathcal{J}^\star(\mathsf{f},\vec{\mathcal{D}})(\vec{v}) & \text{if } (\vec{D},\vec{v}) \sqsubseteq_{\mathcal{U}^m\times W} (\vec{E},\vec{w}) \\ \mathcal{J}(\mathsf{f},\vec{\mathcal{D}})(\vec{v}) & \text{otherwise} \end{cases}$$

Moreover, for all valuations ξ and λ-abstractions $\lambda x : v.\ t$, we let $\mathcal{L}^{\vec{\mathcal{E}},\vec{w}}(\xi, \lambda x.\ t)$ be the function that maps each $v \in [\![v]\!]_{\mathcal{I}_{\mathsf{ty}},\xi}$ to $[\![t]\!]_{\mathcal{J}^{\vec{\mathcal{E}},\vec{w}},\xi[x\mapsto v]}$.

Notice that unlike for $\mathcal{I}^\star$, the interpretation of p by $\mathcal{I}^{\vec{\mathcal{E}},\vec{w}}$ is not given as the interpretation of a term. As a result, the predicate interpreting p is not guaranteed to exist in the interpretation of p's type, and thus $\mathcal{I}^{\vec{\mathcal{E}},\vec{w}}$ might not qualify as an interpretation. For this reason, we call it a "pseudo-interpretation."

In a pseudo-interpretation, quantification is not guaranteed to range over all values denotable by terms. Nevertheless, we can still use $\mathcal{I}^{\vec{\mathcal{E}},\vec{w}}$ like an interpretation to ascertain the truth value of clauses.

We start by proving that our desired result holds for every pseudo-interpretation, by induction. Then we will show that the result extends to $\mathcal{I}^\star$.

Lemma 5.9. *Let N be a clause set and C be a clause contained in N such that C has no variables in common with $N \setminus \{C\}$. Assume C is blocked by L in N. Let $\mathcal{I}$ be a interpretation, and let $\mathcal{I}^{\vec{\mathcal{E}},\vec{w}}$ be the pseudo-interpretation up to some $\vec{\mathcal{E}}$ and $\vec{w}$ and associated with N, C, and L. For every $D \in N \setminus \{C\}$, if $\mathcal{I} \models D$, then $\mathcal{I}^{\vec{\mathcal{E}},\vec{w}} \models D$.*

Proof. The proof is by well-founded induction on $(\vec{\mathcal{E}},\vec{w})$ with respect to $\sqsubset$.

Let $C = L \vee C'$ where $L = (\neg)\,\mathsf{p}\langle\vec{\tau}\rangle\ \vec{s}_n$ (as per condition 1 of Definition 5.1), and let $\mathcal{I}' = \mathcal{I}^{\vec{\mathcal{E}},\vec{w}}$. Assuming that $\mathcal{I} \models D$, we will show that $[\![D]\!]_{\mathcal{I}',\xi}$ is true for any valuation ξ.

Let L' be a literal of D of opposite polarity to L, whose atom is $\mathsf{p}\langle\vec{v}\rangle\ \vec{t}$, and such that, for each j, $[\![v_j]\!]_{\mathcal{I},\xi} = \mathcal{E}_j$, $[\![t_j]\!]_{\mathcal{I},\xi} = w_j$, $[\![L']\!]_{\mathcal{I},\xi}$ is true, and $[\![L']\!]_{\mathcal{I}',\xi}$ is false. Intuitively, L' is a literal whose truth value "flips" from true to false in $\mathcal{I}'$. We distinguish two cases: The case where such a literal L' exists and the case where it does not.

Case where L' exists: We will show that $[\![D]\!]_{\mathcal{I}',\xi} = [\![D]\!]_{\mathcal{I},\xi}$ even if L' has gone from true in $\mathcal{I}$ to false in $\mathcal{I}'$ under ξ. Since $\mathcal{I} \models D$, $[\![D]\!]_{\mathcal{I}',\xi}$ will then be true.

Since $\mathcal{I}'$ "flips" the truth value of L', by construction of $\mathcal{I}^\star$ underlying $\mathcal{I}'$, this must be triggered by C: There must exist a valuation ξ' such that, for every j, $[\![\tau_j]\!]_{\mathcal{I},\xi'} = \mathcal{E}_j$ and $[\![s_j]\!]_{\mathcal{I},\xi'} = w_j$. Let ξ'' be the valuation that coincides with ξ' on C's free variables and with ξ on D's free variables. Let $D = L' \vee D'$.

Since C is blocked by L in N, all binary flat L-resolvents of C with clauses from N are tautologies (by condition 4 of Definition 5.2). In particular, consider the binary flat L-resolvent of C and D of the form $\big(\big(\bigvee_{j=1}^n s_j \not\approx t_j\big) \vee C' \vee D'\big)\sigma$, where σ is the most general unifier of $\vec{\tau} \stackrel{?}{=} \vec{v}$. This binary flat L-resolvent, which must exist by the five conditions of Definition 5.1, is a tautology, and it must be satisfied by $\mathcal{I}'$ under ξ.

By definition of ξ'', $[\![s_j \not\approx t_j]\!]_{\mathcal{I},\xi''}$ must be false, and since the terms $\vec{s}$, $\vec{t}$ do not contain p (by condition 2 of Definition 5.2), $[\![s_j \not\approx t_{ij}]\!]_{\mathcal{I}',\xi''}$ must be false as well. Moreover, by construction of $\mathcal{I}^\star$ underlying $\mathcal{I}'$, the only way for the interpretation of $\mathsf{p}\langle\vec{\tau}\rangle\ \vec{s}_n$ in $\mathcal{I}'$ under ξ'' to differ from that in $\mathcal{I}$ under ξ'' is if $[\![C']\!]_{\mathcal{I},\xi''}$ is false, and since C' contains only occurrences of p of opposite polarity to L (by conditions 2 and 3 of Definition 5.2), $[\![C']\!]_{\mathcal{I}',\xi''}$ must be "even more false" after we "flipped" L to make it true.

Finally, since both $[\![s_j \not\approx t_j]\!]_{\mathcal{I},\xi''}$ and $[\![C']\!]_{\mathcal{I}',\xi''}$ are false and the binary flat L-resolvent of C and D is a tautology, $D'\sigma$ must be true in $\mathcal{I}'$ under any valuation. Since σ is a most general unifier and ξ'' assigns the same semantics to $\vec{\tau}$ and $\vec{v}$, effectively "unifying" them, we also have that $[\![D']\!]_{\mathcal{I}',\xi''}$ is true and hence $[\![D']\!]_{\mathcal{I}',\xi}$ is true. Thus $[\![D]\!]_{\mathcal{I}',\xi}$ is true, as desired.

Case where L' does not exist: Let $\{L'_1, \ldots, L'_m\}$ be the subset of D's p-literals such that each L'_i is of opposite polarity to L, and let $\vec{v'_i}$ and $\vec{t'_i}$ be respectively the type arguments and the term arguments of literal L'_i's predicate symbol p. With each L'_i, associate a tuple $(\vec{\mathcal{E}_i}, \vec{w_i})$ corresponding to the interpretation of $(\vec{v'_i}, \vec{t'_i})$ in $\mathcal{I}$ under ξ. Let $(\vec{\mathcal{E}_0}, \vec{w_0})$ be the $\sqsubset$-maximum tuple among these tuples such that $(\vec{\mathcal{E}_0}, \vec{w_0}) \sqsubset (\vec{\mathcal{E}}, \vec{w})$.

If such a tuple exists, by the induction hypothesis $\mathcal{I}^{\vec{\mathcal{E}_0},\vec{w_0}} \models D$. We will look at the literals of D in turn an argue that they are "more true" in $\mathcal{I}'$ than in $\mathcal{I}^{\vec{\mathcal{E}_0},\vec{w_0}}$, under ξ. This will give us the desired result that $[\![D]\!]_{\mathcal{I}',\xi}$ is true. For the literals L'_i, it is clear by construction of pseudo-interpretations that they have the same semantics in $\mathcal{I}^{\vec{\mathcal{E}_0},\vec{w_0}}$ and $\mathcal{I}'$ under ξ. For the remaining literals, those that contain p contain it with the same polarity as L (by condition 2 of Definition 5.2) and are only made "more true" by $\mathcal{I}'$, and the remaining literals do not contain p and hence have the same semantics in $\mathcal{I}^{\vec{\mathcal{E}_0},\vec{w_0}}$ and $\mathcal{I}'$.

If no tuple $(\vec{\mathcal{E}_0}, \vec{w_0})$ exists, the same reasoning applies as above but using the assumption $\mathcal{I} \models D$ instead of the induction hypothesis. □

Lemma 5.10. *Let N be a clause set and C be a clause contained in N such that C has no variables in common with $N \setminus \{C\}$. Assume C is blocked by L in N. Let $\mathcal{I}$ be an interpretation and $\mathcal{I}^\star$ be the interpretation obtained by flipping the truth value of L. For every $D \in N \setminus \{C\}$, if $\mathcal{I} \models D$, then $\mathcal{I}^\star \models D$.*

Proof. Let $\{L'_1, \ldots, L'_m\}$ be the subset of D's literals such that each L'_i is of opposite polarity to L, and let $\vec{v'_i}$ and $\vec{t'_i}$ be respectively the type arguments and the term arguments of literal L'_i's predicate symbol p. With each L'_i, associate a tuple $(\vec{\mathcal{E}_i}, \vec{w_i})$ corresponding to the interpretation of $(\vec{v'_i}, \vec{t'_i})$ in $\mathcal{I}$ under ξ. Let $(\vec{\mathcal{E}_0}, \vec{w_0})$ be the $\sqsubset$-maximum tuple among these tuples such that $(\vec{\mathcal{E}_0}, \vec{w_0}) \sqsubset (\vec{\mathcal{E}}, \vec{w})$.

If such a tuple exists, we invoke Lemma 5.9 to obtain $\mathcal{I}^{\vec{\mathcal{E}_0},\vec{w_0}} \models D$. To obtain $\mathcal{I}^\star \models D$, as desired, we make an argument similar to the second case in the proof of Lemma 5.9: The literals L'_i keep their truth value beyond $\vec{\mathcal{E}_0}, \vec{w_0}$, p-literals of the same polarity as L only become "more true," and non-p-literals keep their truth value.

If no tuple $(\vec{\mathcal{E}_0}, \vec{w_0})$ exists, we make the same argument but starting from $\mathcal{I} \models D$ instead of $\mathcal{I}^{\vec{\mathcal{E}_0},\vec{w_0}} \models D$. □

Lemma 5.11. *Let N be a clause set and C be a clause contained in N. If C is blocked by a literal in N, then $N \setminus \{C\}$ is satisfiable if and only if N is satisfiable.*

Proof. The "if" direction is trivial. For the other direction, let N' consist of all clauses from $N \setminus \{C\}$ with their type and term variables renamed so that they share no variables with C. Let $\mathcal{I}$ be a model of N'. By Lemma 5.10, $\mathcal{I}^\star$ is a model of each $D \in N'$.

We also need to show that $\mathcal{I}^\star$ is a model of C. Specifically, we must show that $[\![C]\!]_{\mathcal{I}^\star,\xi}$ is true for any valuation ξ. Let $C = L \vee C'$. If $[\![C']\!]_{\mathcal{I}^\star,\xi}$ is true, we are done. Otherwise, first suppose L is positive. Then ξ provides the necessary witnesses for the $\exists$ quantifier in the definition of φ in Definition 5.6, making the interpretation of p by $\mathcal{I}^\star$ true in that case, as in

the first part of Example 5.7. Hence $[\![L]\!]_{\mathcal{I}^\star,\xi}$ is true, and thus $[\![C]\!]_{\mathcal{I}^\star,\xi}$ is true. Next, suppose L is negative. Then ξ provides a counterexample to the $\forall$ quantifier in the definition of φ, making the interpretation of p by $\mathcal{I}^\star$ false in that case, as in the second part of Example 5.7. Hence $[\![L]\!]_{\mathcal{I}^\star,\xi}$ is true, and thus $[\![C]\!]_{\mathcal{I}^\star,\xi}$ is true.

Since $\mathcal{I}^\star \models N' \cup \{C\}$, we have that N is satisfiable. □

Definition 5.12. Given a finite clause set, *blocked clause elimination* (BCE) repeatedly removes blocked clauses until no such clauses remain.

The procedure is confluent and hence yields a unique result. This is easy to see because removing a blocked clause will only make more clauses blocked; it can never "unblock" a clause.

Theorem 5.13. *The result of applying BCE to a clause set N is satisfiable if and only if N is satisfiable.*

Proof. This follows by iteration of Lemma 5.11. □

6. Quasipure Literal Elimination

Pure literal elimination (PLE) is one of the simplest optimizations implemented in SAT solvers. It is a special case of variable elimination [SP04, CS00]: If a given variable always occurs with the same polarity in a problem, the solver can assign it that polarity without loss of generality, making all the clauses that contain it tautologies. PLE consists of recursively deleting all such clauses. PLE's generalization to first-order logic considers literals $(\neg)\ \mathsf{p}(\vec{s})$, where the arguments $\vec{s}$ are ignored by the analysis; only the polarity is considered. The same idea carries over to higher-order logic.

Example 6.1. Consider the clause set $N = \{\mathsf{p}\,x \vee \mathsf{q}\,\mathsf{a}\,x,\ \mathsf{p}\,(\mathsf{f}\,x),\ \neg\,\mathsf{q}\,\mathsf{a}\,\mathsf{a}\}$. Since p occurs only positively in N, PLE removes the two first clauses. At that point, q occurs only negatively in the remaining singleton clause set and can be removed as well. The result is the empty set, which is obviously satisfiable, indicating that N is satisfiable. As model of N, we can take an interpretation $\mathcal{I}$ that makes all p-literals true and all q-literals false.

Example 6.2. Consider the clause set $N' = \{\mathsf{p}\,\mathsf{a},\ \neg\,\mathsf{p}\,x \vee \mathsf{p}\,(\mathsf{f}\,x)\}$. PLE does not apply because p occurs with both polarities. Yet we notice that p occurs positively in both clauses, and hence that the same reasoning as in Example 6.1 applies: We can satisfy both clauses by making p-literals true. Using $\mathcal{I}$ from Example 6.1, we have $\mathcal{I} \models N'$.

Example 6.3. Consider the clause set $N'' = \{\mathsf{p}\mathsf{a},\ \mathsf{q}x \vee \mathsf{p}(\mathsf{f}x),\ \neg\,\mathsf{q}\,(\mathsf{f}\,\mathsf{a}),\ \neg\,\mathsf{p}\,x \vee \neg\,\mathsf{q}\,(\mathsf{h}\,\mathsf{p}\,(\mathsf{p}\,\mathsf{b}))\}$. PLE does not apply because p and q occur with both polarities. In addition, p also occurs unapplied and deep within a term. Yet each clause contains either a positive p-literal or a negative q-literal. Thus $\mathcal{I} \models N''$, where $\mathcal{I}$ is as in Example 6.1. The additional literals are harmless.

Examples 6.2 and 6.3 suggest that pure literals are a needlessly restrictive criterion in first- and higher-order logic. We propose a generalization to "quasipure literals."

Definition 6.4. A *polarity map* is a function that maps each predicate symbol in Σ to a polarity ($+$ or $-$). A set P of predicate symbols is *quasipure* in a clause set N with a polarity map m if for every symbol $\mathsf{p} \in P$, for every clause in N that contains an element

of P, there exists a predicate symbol $\mathsf{q} \in P$ such that the clause contains a q-literal with polarity m_{q}. The set P is *quasipure* in N if there exists a polarity map m such that P is quasipure in N with m.

In Example 6.1, $\{\mathsf{p}, \mathsf{q}\}$ is quasipure in N with $m_{\mathsf{p}} = +$ and $m_{\mathsf{q}} = -$. In Example 6.2, $\{\mathsf{p}\}$ is quasipure in N' with $m_{\mathsf{p}} = +$. In Example 6.3, $\{\mathsf{p}, \mathsf{q}\}$ is quasipure in N'' with $m_{\mathsf{p}} = +$ and $m_{\mathsf{q}} = -$. For this last example, it is crucial to consider p and q together; neither of the singletons $\{\mathsf{p}\}$ and $\{\mathsf{q}\}$ is quasipure in N''.

Definition 6.5. A predicate symbol p is *quasipure* in a clause set N with polarity $s \in \{+, -\}$ if there exists a set P of predicate symbols with $\mathsf{p} \in P$ and a polarity map m such that $m_{\mathsf{p}} = s$ and P is quasipure in N with m. The symbol p is *quasipure* in N if there exists a polarity $m_{\mathsf{p}} \in \{+, -\}$ such that p is quasipure in N with m_{p}. A literal $L = (\neg)\ \mathsf{p}\ \ldots$ is *quasipure* in N if p is quasipure in N with L's polarity.

Notice that a predicate symbol that does not occur in a clause set is trivially quasipure in that clause set.

Deleting a clause containing a quasipure literal might create new opportunities for quasipure literal elimination, but it never ruins existing ones. Therefore, the following nondeterministic procedure is confluent and yields a unique result:

Definition 6.6. Given a finite clause set, *quasipure literal elimination* (QLE) repeatedly removes clauses containing quasipure predicate symbols until no such symbols remain.

Although QLE is defined by iteration, it is always possible to eliminate all quasipure predicate symbols at the same time:

Lemma 6.7. *Let N be a finite clause set and let N' be the result of QLE. Then there exists a predicate symbol set P and a polarity map m such that P is quasipure in N and for every clause in $N \setminus N'$ there exists a predicate symbol $\mathsf{q} \in P$ such that the clause contains a q-literal with polarity m_{q}.*

Proof. The iterative process defining QLE gives rise to a finite sequence $(P_1, m_1), \ldots, (P_n, m_n)$ of predicate symbol sets and polarity maps. Without loss of generality, we assume that each P_j does not contain predicate symbols that do not occur in the clause set at iteration j. Then the sets P_j are clearly mutually disjoint and we can take $P = P_1 \cup \cdots \cup P_n$ as the desired witness. As for the polarity map m, we associate each $\mathsf{p} \in P_j$ with $m_j(\mathsf{p})$. □

The key property of QLE is that it preserves unsatisfiability:

Lemma 6.8. *Let $C \in N$ be a clause containing a quasipure literal. If $N \setminus \{C\}$ is satisfiable, then N is satisfiable.*

Proof. Let $\mathcal{I}$ be a model of $N \setminus \{C\}$. Let P be the set of predicate symbols and m the polarity map whose existence is guaranteed by Lemma 6.7. Let $N_0 \subseteq N \setminus \{C\}$ be the result of applying QLE on N. Clearly, N_0 contains no occurrences of the symbols in P. Define $\mathcal{I}'$ based on $\mathcal{I}$ by redefining the semantics of each monomorphic instance $\mathsf{p}\langle\vec{\tau}\rangle : \upsilon_1 \to \cdots \to \upsilon_n \to o$ of symbol p such that $m_{\mathsf{p}} = +$ or $m_{\mathsf{p}} = -$: If $m_{\mathsf{p}} = +$, interpret $\mathsf{p}\langle\vec{\tau}\rangle$ as the predicate that is uniformly true; otherwise, interpret $\mathsf{p}\langle\vec{\tau}\rangle$ as the predicate that is uniformly false. By the comprehension principle, both of these predicates are guaranteed to exist in the interpretation of the type $\upsilon_1 \to \cdots \to \upsilon_n \to o$. Now $\mathcal{I}'$ coincides with $\mathcal{I}$, since N_0 contains no P symbols, and thus $\mathcal{I}'$ is a model of N_0. In addition, $\mathcal{I}'$ is a model of N, because each clause in $N \setminus N_0$ contains a quasipure literal, which is satisfied by $\mathcal{I}'$. □

Definition 6.6 suggests a naive, nondeterministic procedure for discovering and eliminating quasipure predicate symbols: Choose a predicate symbol p and a polarity m_{p}, and take $P = \{\mathsf{p}\}$. If the predicate symbol occurs with the wrong polarity in a clause $(\neg)\,\mathsf{p} \ldots \vee (\neg)\,\mathsf{q}_1 \ldots \vee \cdots \vee (\neg)\,\mathsf{q}_n \ldots \vee C$, try to extend the set P with one of the $(\neg)\,\mathsf{q}_i$'s and the polarity map m accordingly, and continue recursively with q_i. In Section 7, we will see a more efficient approach based on a SAT encoding.

7. Implementation

We implemented the elimination techniques described in Sections 3 to 6 in the Zipperposition prover. For HLBE, PE, and BCE, which had been studied by Vukmirović et al., we could directly adapt their code [VBH21a, Section VI]. The data structure and algorithms they described and implemented could be generalized to handle polymorphic higher-order logic. For QLE, we developed our own code.

Zipperposition is a higher-order prover that implements the λ-superposition calculus [BBTV21], a generalization of standard superposition to classical rank-1 polymorphic higher-order logic—the logic described in Section 2. The prover is highly competitive: It won in the higher-order theorem division of the CADE ATP Systems Competition (CASC) [SD22] in 2020, 2021, and 2022.

By default, Zipperposition *immediately clausifies* the initial problem as much as possible; then it optionally invokes preprocessing elimination techniques. If inprocessing is enabled, the prover also invokes the elimination techniques at regular intervals from within the saturation loop. Immediate clausification performs well in practice, but an even more successful strategy is to *delay clausification*, interleaving clausification with superposition-style calculus rules [VBB+21, Section 4]. Then it makes little sense to apply preprocessing elimination techniques; inprocessing seems more appropriate.

7.1. Hidden-Literal-Based Elimination.

HLBE relies on matching. In our setting, it needs to consider type variables and higher-order terms. Our implementation uses a restricted but efficient form of higher-order matching, which recognizes $\lambda y.\, y\,\mathsf{a}$ as an instance of $\lambda y.\, y\,x$, but not a as an instance of $y\,\mathsf{a}$ (with $y := \lambda x.\, x$). The same matching algorithm is used in Zipperposition to efficiently recognize subsumed clauses. This weaker matching reduces the applicability of HLBE, but it does not compromise its soundness.

7.2. Predicate Elimination.

Compared with HLBE, more work was needed to make predicate elimination cope with polymorphism and higher-order logic. For polymorphism, the previous code simply did not consider polymorphic predicates as predicates. Predicates needed to have a type declaration of the form $\tau_1 \times \cdots \times \tau_n \to o$. Now, predicate symbols can have type declarations of the form $\Pi\vec{\alpha}.\tau_1 \to \cdots \to \tau_n \to o$ as well as $\Pi\vec{\alpha}.\tau_1 \to \cdots \to \tau_n \to \alpha_i$; via instantiation, α_i can become o or $\cdots \to o$.

For SPE, the type arguments of eliminated predicate symbols must be unified, as per Definition 4.4. For DPE, the type arguments must be distinct type variables. In addition, for DPE, we must check that the predicate is polymorphism-safe.

Adding support for higher-order logic required changing the definition of singular predicates, which is used by both SPE and DPE, to check that the predicate symbol to eliminate does not occur deep in the clauses that define the symbol. In addition, for DPE, we

need to synthesize a λ-abstraction to replace any deep occurrences of the predicate outside the definition set.

7.3. **Blocked Clause Elimination.** Adding support for polymorphism to BCE was straightforward: We simply ensured that type arguments are unified when computing flat resolvents and added a polymorphism-safety check.

To support higher-order logic, we added a check that the p-literal on which the clause is resolved in the only p-literal of that polarity (corresponding to condition 3 of Definition 5.2). We also disabled the code that computed n-ary resolvents for $n > 2$. Finally, we added code to compute the list of all deep predicate symbols q and made sure clauses are never blocked on a q-literal. These mechanisms come into play only if some higher-order construct is detected in the input problem; otherwise, the first-order formulation of BCE is used.

In the presence of equality in the logic, BCE relies on a congruence closure algorithm to detect valid clauses [KSS+17, Section 6.1]. In our implementation, we rely on a first-order congruence closure algorithm, which can handle higher-order constructs but does not take advantage of them. For example, resolving $\mathsf{p}\,\mathsf{a} \vee \neg\,\mathsf{q}\,\mathsf{a}$ against $\neg\,\mathsf{p}\,\mathsf{b} \vee \mathsf{q}\,\mathsf{b}$ on the p-literal yields the clause $\mathsf{a} \not\approx \mathsf{b} \vee \neg\,\mathsf{q}\,\mathsf{a} \vee \mathsf{q}\,\mathsf{b}$. Our congruence closure algorithm can detect the validity of such a clause. On the other hand, because the algorithm views λ-abstractions as black boxes, it fails to recognize $\mathsf{a} \not\approx \mathsf{b} \vee \neg\,\mathsf{q}\,(\lambda x.\, x\,\mathsf{a}) \vee \mathsf{q}\,(\lambda x.\, x\,\mathsf{b})$ as valid.

7.4. **Quasipure Literal Elimination.** A simple and efficient implementation of quasipure literal elimination uses a SAT solver. Let N be a finite clause set. Without loss of generality, we can assume that Σ is finite. The signature of the generated SAT problem consists of the variables $\mathsf{p}^+, \mathsf{p}^-$ for each predicate symbol p in Σ, where p^s means "p (possibly together with other predicate symbols) is quasipure with polarity s." The SAT problem consists of the following clauses:

(1) For each clause in N containing n predicate literals, headed by $\mathsf{q}_1, \ldots, \mathsf{q}_n$ and with respective polarities $s_1, \ldots, s_n$, generate n clauses of the form

$$\mathsf{q}_1^{s_1} \vee \cdots \vee \mathsf{q}_{j-1}^{s_{j-1}} \vee \neg\,\mathsf{q}_i^{-s_j} \vee \mathsf{q}_{j+1}^{s_{j+1}} \vee \cdots \vee \mathsf{q}_n^{s_n}$$

where $-s$ flips the polarity s. Such clauses ensure that whenever a literal $(\neg)\ \mathsf{q}_i \ldots$ has the wrong polarity according to the current variable assignment, one of the other predicate literals must be quasipure.

(2) For each clause in N containing n predicate literals, headed by $\mathsf{q}_1, \ldots, \mathsf{q}_n$ and with respective polarities $s_1, \ldots, s_n$ and containing a deep occurrence of p (in an argument to a q_j or in a functional literal), generate the two clauses

$$\neg\,\mathsf{p}^+ \vee \mathsf{q}_1^{s_1} \vee \cdots \vee \mathsf{q}_n^{s_n} \qquad\qquad \neg\,\mathsf{p}^- \vee \mathsf{q}_1^{s_1} \vee \cdots \vee \mathsf{q}_n^{s_n}$$

These clauses ensures that whenever p occurs deep and is nonetheless considered quasipure, one of the predicate literals must be quasipure.

(3) For each predicate symbol p in Σ, generate the clause $\neg\,\mathsf{p}^+ \vee \neg\,\mathsf{p}^-$. It ensures that a single polarity is assigned to a quasipure predicate symbol.

(4) Generate the clause $\mathsf{p}_1^+ \vee \mathsf{p}_1^- \vee \cdots \vee \mathsf{p}_n^+ \vee \mathsf{p}_n^-$, where $\{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$ are the predicate symbols in Σ. It tells the SAT solver to look for a nontrivial solution, in which at least one predicate symbol is quasipure.

From a satisfying assignment, we can easily read off a predicate symbol set and a polarity map. The process can be iterated until we reach a maximal solution, at which point the SAT solver returns a verdict of "unsatisfiable."

So far, we have been unable to prove that this problem is **NP**-complete. Given the nondeterministic nature of the naive procedure, we suspect that it is, but we have not found a reduction from 3-SAT. Thus, it is unclear whether our use of a SAT solver is fully satisfactory from a theoretical point of view, even though it works well in practice.

8. Evaluation

We evaluate the techniques presented above by running Zipperposition [BBTV21] on various benchmarks in various configurations. We consider five benchmark sets:

- *T0*: a random subset of 1000 higher-order monomorphic (TH0) problems from the TPTP [Sut17] version 8.0.0;
- *T1*: a random subset of 1000 higher-order polymorphic (TH1) problems from the TPTP version 8.0.0;
- *S0*: a random subset of 1000 higher-order monomorphic (TH0) problems from the Sledgehammer-generated Seventeen benchmark suite [DVBW22] in the base configuration, in the language fragment called $\mathrm{TH0}^-$ in the Seventeen paper;
- *S1*: a random subset of 1000 higher-order polymorphic (TH1) problems from the Seventeen benchmark suite in the base configuration, in the language fragment called $\mathrm{TH1}^-$ in the Seventeen paper;
- *F*: a predefined set of 1000 first-order untyped (CNF and FOF) problems from the TPTP.

The first four benchmark sets are used to determine how much our techniques can help. Among these, T1 and S1 exercise polymorphism. As for F, it is included for comparison; it shows how well the techniques work on first-order problems. T0, T1, and F (and probably S0 and S1 as well) contain some unprovable problems, which can be used to check soundness of the techniques.

We consider 12 Zipperposition configurations, all derived from the portfolio of time slices that was used at the 2022 edition of CASC. This portfolio does not use any of our techniques. Even on first-order problem, it applies the higher-order λ-superposition calculus. The 12 configurations are as follows:

- *None*: the baseline, corresponding to the portfolio used at CASC;
- *X-pre*: the baseline modified to use technique $X \in \{\mathrm{PE}, \mathrm{BCE}, \mathrm{PLE}, \mathrm{QLE}\}$ as a preprocessor in all the time slices;
- *X-in*: the baseline modified to use technique $X \in \{\mathrm{HLBE}, \mathrm{PE}, \mathrm{BCE}, \mathrm{PLE}, \mathrm{QLE}\}$ as an inprocessor in all the time slices;
- *All-pre*: the baseline modified to use all of PE, BCE, and QLE as preprocessors in all the time slices;
- *All-in*: the baseline modified to use all of HLBE, PE, BCE, and QLE as inprocessors in all the time slices.

In addition, we define the virtual configuration *Union*, consisting of the virtual portfolio of all other 12 configurations. All problems that are solved in at least one of the 12 configurations are considered solved by Union, and only those.

	None	HLBE	PE		BCE		PLE		QLE		All		Union
		in	pre	in	pre	in	pre	in	pre	in	pre	in	
T0	711	713	713	709	713	**714**	710	713	711	705	711	706	722
T1	338	334	339	340	**341**	339	337	**341**	337	336	338	336	354
S0	**575**	574	574	572	**575**	574	574	**575**	**575**	574	572	**575**	589
S1	349	343	351	350	347	348	352	352	352	353	**354**	344	366
F	504	513	512	510	509	507	513	505	508	505	517	**519**	545
Total	2477	2477	2489	2481	2485	2482	2486	2486	2483	2473	**2492**	2480	2576

Figure 1: Number of solved problems per benchmark set and configuration

The experiments were carried out on StarExec Miami [SST14] servers equipped with Intel Xeon E5-2620 v4 CPUs clocked at 2.10 GHz. We used CPU and wallclock time limits of 120 s. The raw results are available online.[2]

Figure 1 reports how many problems were solved for each combination of benchmark set and configuration. The last row of the table presents the total of the five rows above it. Bold singles out the best configuration (other than Union) for each benchmark set.

The results are sobering. We see substantial gains on first-order benchmarks (the F row of the table), but the gains are much more modest, if actually present, on higher-order benchmarks. A possible explanation is that there is less clausal structure in a higher-order problem. Most of Zipperposition's time slices clausify the problem lazily, meaning that little information is visible to our techniques, especially when used as preprocessors. Another possible explanation might be that the higher-order TPTP and Seventeen benchmarks look quite different from the first-order TPTP benchmarks, and our techniques are less applicable. For example, the higher-order TPTP problems tend to be much smaller than their first-order counterparts.

The picture is more positive if we look at the Union column of the table. Clearly, in a portfolio setting, with enough time, the new techniques can make a useful contribution.

We also notice that inprocessing often performs less well than preprocessing. This corroborates the findings of Vukmirović et al. [VBH21a]. An explanation might be the heavy cost of running the techniques multiple time, during proof search. In addition, PE, BCE, PLE, and QLE rely on a global analysis of the clause set and tend to become less applicable as the clause set grows.

Finally, we see that pure and quasipure literal elimination help, especially on first-order problems. Unexpectedly, PLE generally outperforms the more general QLE.

9. Conclusion

We presented four SAT-inspired techniques for transforming higher-order problems with the aim of making them more amenable to automatic proof search. Three of the techniques (HLBE, PE, and BCE) had been previously generalized to first-order logic; we now generalized them further to higher-order logic. The fourth technique (QLE) is new.

On the theoretical side, we showed that the techniques preserve satisfiability and unsatisfiability of problems with respect to Henkin semantics. On the practical side, we implemented the techniques in the higher-order prover Zipperposition. Regrettably, the

[2] https://zenodo.org/record/6997515

techniques did not perform as well on higher-order problems as they do on first-order problems. This could be due to the nature of the benchmark sets. More research is needed.

Acknowledgment. Alexander Bentkamp gave us some advice about general interpretations. This 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). The 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).

References

[BBTV21] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirović. Superposition for full higher-order logic. In André Platzer and Geoff Sutcliffe, editors, *CADE-28*, volume 12699 of *LNCS*, pages 396–412. Springer, 2021.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[BR20] Ahmed Bhayat and Giles Reger. A combinator-based superposition calculus for higher-order logic. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *IJCAR 2020, Part I*, volume 12166 of *LNCS*, pages 278–296. Springer, 2020.

[CS00] Philippe Chatalic and Laurent Simon. ZRES: The old Davis–Putnam procedure meets ZBDD. In David A. McAllester, editor, *CADE-18*, volume 1831 of *LNCS*, pages 449–454. Springer, 2000.

[DVBW22] Martin Desharnais, Petar Vukmirovic, Jasmin Blanchette, and Makarius Wenzel. Seventeen provers under the hammer. In June Andronick and Leonardo de Moura, editors, *ITP 2022*, volume 237 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2022.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT 2005*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.

[Fit02] Melvin Fitting. *Types, Tableaus, and Gödel's God*. Kluwer, 2002.

[GO92] Dov M. Gabbay and Hans Jürgen Ohlbach. Quantifier elimination in second-order predicate logic. In Bernhard Nebel, Charles Rich, and William R. Swartout, editors, *KR '92*, pages 425–435. Morgan Kaufmann, 1992.

[HJB11] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Efficient CNF simplification based on binary implication graphs. In Karem A. Sakallah and Laurent Simon, editors, *SAT 2011*, volume 6695 of *LNCS*, pages 201–215. Springer, 2011.

[KK16] Zurab Khasidashvili and Konstantin Korovin. Predicate elimination for preprocessing in first-order theorem proving. In Nadia Creignou and Daniel Le Berre, editors, *SAT 2016*, volume 9710 of *LNCS*, pages 361–372. Springer, 2016.

[KSR16] Cezary Kaliszyk, Geoff Sutcliffe, and Florian Rabe. TH1: The TPTP typed higher-order form with rank-1 polymorphism. In Pascal Fontaine, Stephan Schulz, and Josef Urban, editors, *PAAR-2016*, volume 1635 of *CEUR Workshop Proceedings*, pages 41–55. CEUR-WS.org, 2016.

[KSS+17] Benjamin Kiesl, Martin Suda, Martina Seidl, Hans Tompits, and Armin Biere. Blocked clauses in first-order logic. In Thomas Eiter and David Sands, editors, *LPAR-21*, volume 46 of *EPiC Series in Computing*, pages 31–48. EasyChair, 2017.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Ohl96] Hans Jürgen Ohlbach. SCAN—elimination of predicate quantifiers. In Michael A. McRobbie and John K. Slaney, editors, *CADE-13*, volume 1104 of *LNCS*, pages 161–165. Springer, 1996.

[SB18] Alexander Steen and Christoph Benzmüller. The higher-order prover Leo-III. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *IJCAR 2018*, volume 10900 of *LNCS*, pages 108–116. Springer, 2018.

[SD22] Geoff Sutcliffe and Martin Desharnais. The CADE-28 Automated Theorem Proving System Competition—CASC-28. *AI Commun.*, 34(4):259–276, 2022.

[SP04] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. In Holger H. Hoos and David G. Mitchell, editors, *SAT 2004*, volume 3542 of *LNCS*, pages 276–291. Springer, 2004.

[SST14] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *IJCAR 2014*, volume 8562 of *LNCS*, pages 367–373. Springer, 2014.

[Sut17] Geoff Sutcliffe. The TPTP problem library and associated infrastructure—from CNF to TH0, TPTP v6.4.0. *J. Autom. Reason.*, 59(4):483–502, 2017.

[VBB+21] Petar Vukmirovic, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. Making higher-order superposition work. In André Platzer and Geoff Sutcliffe, editors, *CADE-28*, volume 12699 of *LNCS*, pages 415–432. Springer, 2021.

[VBH21a] Petar Vukmirović, Jasmin Blanchette, and Marijn J.H. Heule. SAT-inspired eliminations for superposition. In *FMCAD 2021*, pages 231–240. IEEE, 2021.

[VBH21b] Petar Vukmirović, Jasmin Blanchette, and Marijn J.H. Heule. SAT-inspired eliminations for superposition. Technical report, 2021. URL: https://matryoshka-project.github.io/pubs/satelimsup_report.pdf.

[VBS] Petar Vukmirović, Jasmin Blanchette, and Stephan Schulz. Extending a high-performance prover to higher-order logic. Unpublished manuscript. URL: https://matryoshka-project.github.io/pubs/lambdae.pdf.