FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Preprocessing in Higher-order Reasoning

Learning from QBF Solving

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Hans-Jörg Schurr
Matrikelnummer 0925891

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr.phil. Alexander Leitsch
Mitwirkung: Priv.-Doz. Dr.-Ing. Christoph Benzmüller

Wien, 17. August 2017

Hans-Jörg Schurr	Alexander Leitsch

Erklärung zur Verfassung der Arbeit

Hans-Jörg Schurr
Hindenburgdamm 57a
12203 Berlin

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 17. August 2017

Hans-Jörg Schurr

Diploma Thesis

Preprocessing in Higher-order Reasoning

Learning from QBF solving

Hans-Jörg Schurr

17. August 2017

Written during a visit to the FU Berlin supported by Priv.Doz. Dr.Ing. Christoph Benzmüller

Advised at TU Wien by Univ.Prof. Dr.Phil. Alexander Leitsch

Kurzfassung

In dieser Diplomarbeit präsentieren wir mehrere Algorithmen zur Vorverarbeitung von Formeln der höherstufigen Logik. Diese Logik erlaubt, neben der Quantifizierung über Individuen, auch die Quantifizierung über Funktionen und Relationen. Wir beschäftigen uns im speziellen mit dem Problem des automatischen Beweisens. Ein automatischer Beweiser für die Logik höherer Stufe ist das Leo-III System. Im Rahmen der vorliegenden Arbeit, wurde Leo-III um mehrere Algorithmen, welche auf Verfahren zur Vorverarbeitung von quantifizierte Boolsche Formeln basieren, erweitert. Quantifizierte Boolsche Formeln sind eine Erweiterung der Aussagenlogik und erlauben zusätzlich die Quantifizierung über Wahrheitswerte. Diese Formeln sind ein Fragment der höherstufigen Logik.

In den ersten zwei Kapiteln der vorliegenden Arbeit stellen wir diese zwei Logiken ausführlich vor. Viele moderne Systeme zum Lösen von quantifizierten Boolschen Formeln setzen auf einen Vorverarbeitungsschritt. Ein solcher Schritt produziert ein äquivalentes, aber potentiell einfacheres Problem. Wir untersuchten, ob es möglich ist einige dieser Algorithmen auf die Logik höherer Stufe zu adaptieren. Deshalb präsentieren wir zuerst einige dieser Techniken und stellen dann die folgenden vier Techniken für die Logik höherer Stufe vor:

Universal reduction erlaubt das Entfernen von Literalen aus Klauseln, wenn diese universell quantifiziert sind und in keinem anderen Literal der Klausel auftauchen. *Constant extraction* verwendet einen SAT Löser um Literale zu finden, welche notwendigerweise von der Formel impliziert sind, und fügt diese als neue Klausel zum Problem hinzu. *Blocked clause elimination* ermöglicht das Entfernen von Klausel wenn alle Resolventen bezüglich eines Literals tautologisch sind. Unsere Adaption verwendet Pattern-Literale und funktioniert mit Formeln ohne Gleichheit. *First-order re-encoding* verpackt die Literale eines Problems in eine spezielle Relation um die primitive Substitution zu verzögern.

Alle vier Techniken wurden für den Beweiser Leo-III implementiert. Weiters wurden zwei Werkzeuge programmiert, welche quantifizierte Boolsche Formeln in Formeln der höherstufigen Logik übersetzen. Leo-III konnte aus nur wenige dieser Probleme lösen. Die Algorithmen wurden deshalb mit Problemen aus der TPTP Bibliothek untersucht.

Abstract

In this thesis we present some preprocessing techniques for theorem proving in higher-order logic which are based on preprocessing techniques for quantified Boolean formulas. Higher-Order Logic (HOL) is a language which extends the well-known first-order logic with support for quantification over predicates and functions. In particular, we are interested in the development of automatic theorem provers for this logic. Automatic theorem provers attempt to find proofs for HOL formulas without further user input. One such system is Leo-III.

Quantified Boolean Formulas (QBF), on the other hand, are an extension of propositional logic with explicit quantification over truth values. The validity of QBFs is, in contrast to HOL, decidable. Since HOL also supports quantification over truth values it is possible to translate every QBF into an equivalent HOL formula. We also discuss DQBF, an extension of QBF.

Most modern QBF solving systems apply a preprocessing step before they start solving the input problem. Often this is done by invoking an external preprocessing tool. The output of the preprocessing tool is a potentially easier, but equivalent problem. We investigated whether it is possible to adapt some of the techniques used by QBF preprocessing systems to HOL. Towards this end, we give an overview of common QBF preprocessing techniques and present four algorithms for HOL.

Universal reduction allows the removal of literals from clauses if they are universally quantified and appear in no other literal of the clause. *Constant extraction* uses a SAT solver to find literals necessary implied by the problem and adds them as a unit clause. *Blocked clause elimination* allows the removal of clauses if all resolvents on a literal are tautological. Our adaption uses pattern literals and works for problems without equality. Finally, *first-order re-encoding* wraps literals in a fresh proposition to delay primitive substitution.

All four preprocessing techniques were implemented in the Leo-III theorem prover. To evaluate the performance of Leo-III and Leo-III augmented with our techniques on QBF and DQBF problems, we developed two flexible tools which translate those problems into HOL. Unfortunately, Leo-III could only solve a few of these. Hence, we evaluate the impact of our preprocessing techniques on HOL problems instead.

Contents

1 Introduction

The field of automatic reasoning is concerned with developing computer programs which perform formal logical reasoning without additional user input. Such reasoning might be the construction of proofs, checking proofs for their correctness, or searching for models of logical formulas.

This thesis connects two areas in automatic reasoning: Higher-order theorem proving and QBF solving. Higher-order logic is an expressive logic which allows quantification over functions and relations in addition to quantification over individuals. Quantified Boolean Formulas (QBF) are an extension of classical propositional formulas with quantification over Boolean values. The set of QBFs forms a fragment of higher-order formulas. Recent tools for solving QBFs rely heavily on preprocessing techniques (see section 3.2) and the question arises: Can some of those preprocessing techniques be lifted to higher-order theorem proving? This question is the starting point for this thesis.

The first two chapters introduces the background used in the subsequent chapters.

Since we aim to improve higher-order theorem proving, we first describe higher-order logic, including its syntax and semantics. Furthermore, we give an overview of software systems related to higher-order theorem proving. Our description will focus on Leo-III, the system which was extended by our preprocessing techniques.

Subsection 2.3.2 describes the architecture of the Leo-III prover and the used proof calculus. Then, subsection 2.3.3 describes the TPTP project. The TPTP infrastructure provides a standard format to represent logic problems and proofs. Furthermore, the TPTP library contains an extensive set of logical problems.

In chapter 3, we describe quantified Boolean formulas. Their syntax and semantics is much simpler than syntax and semantics of full higher-order logic. Afterwards the design of modern QBF solvers is discussed in section 3.2. We focus our discussion on systems which performed well in a recent QBF solving competition. Most of those systems relied on preprocessing.

Before giving an overview of preprocessing techniques used in QBF solving at the end of this chapter, we introduce Dependently Quantified Boolean Formulas (DQBF), a generalization of QBFs.

In the fourth chapter, we propose four preprocessing techniques for higher-order theorem proving. We start with lifting *universal reduction* to HOL. Universal reduction is an integral part of resolution for QBF and has a direct correspondence in higher-order logic.

Then we introduce a technique which uses a SAT solver to find constant literals. Constant literals are added to the input problem as new unit clauses without changing its satisfiability status. Furthermore, transitivity constraints induced by the equalities appearing in the input problem enrich the SAT problem.

The third proposed preprocessing technique is *blocked clause elimination*. This is a successful and important technique in QBF preprocessing. Lifting blocked clause elimination to higher-order logic is possible, but results in additional challenges. We present an adaptation of blocked clause elimination to higher-order logic without equality which utilizes the pattern fragment of higher-order logic.

Finally, we investigate *first-order re-encoding*. This technique is not directly inspired by a QBF preprocessing technique, but imitates the encoding of QBFs into first-order logic. By doing so, application of the primitive substitution rule can be delayed. This rule guesses instantiations and therefore induces non-determinism into the search procedure. We describe primitive substitution in section 2.3.1.

In chapter 5 we describe the implementation of the preprocessing techniques presented above. We start with an overview of the challenges encountered while implementing the preprocessing techniques. Then we discuss a set of tools which was developed to encode QBFs and DQBFs as higher-order problems in the TPTP format. Documentation for the tools can be found in section 5.2. Afterwards, the results of evaluating the implemented techniques are presented.

Finally, chapter 6 puts the work done in this thesis in a wider context by presenting related work and suggesting future work.

2 Higher-order Logic

In this section we introduce higher-order logic. We start with an informal discussion and then present its formal syntax and semantics. For the sake of brevity we will use the abbreviation HOL for higher-order logic from now on. In fact there are many higher-order logics and the abbreviation HOL will be used for a specific incarnation of Church's type theory as discussed below.

Higher-order logic is a generalization of second-order logic, which itself is, not surprisingly, a generalization of first-order logic. The Stanford Encyclopedia of Philosophy introduces second-order logic in the following way:

> Second-order logic is an extension of first-order logic where, in addition to quantifiers such as "for every object (in the universe of discourse)," one has quantifiers such as "for every *property* of objects (in the universe of discourse)." (Enderton 2015)

Second-order logic is a genuine extension of first-order logic. Consider, for example, an axiomatization of the natural numbers. One well-known property of the natural numbers is *well-ordering*: Every set of natural numbers has a smallest member. If P is a predicate symbol with one argument, then the first-order sentence

$$\exists x.\, Px \to \exists x.\, (Px \wedge \forall y.\, (Py \to (y = x \vee x < y)))$$

expresses the idea, that if P is true for at least one element, then there is an element for which P is true and all other elements, for which P is true, are bigger. Then, to express the well-ordering property, we need to speak about *all* properties (sets) P. This results in the second-order sentence

$$\forall P.\, (\exists x.\, Px \to \exists x.\, (Px \wedge \forall y.\, (Py \to (y = x \vee x < y)))).$$

In first-order logic, however it is always possible to find models of Peano arithmetic which are not isomorphic to the natural numbers. Those *non-standard models* are not well-ordered. Thus, it is not possible to express the well-ordering of the natural numbers as a first-order sentence (see Enderton 2015).

After allowing quantification over properties of individuals, it is just natural to allow quantification over properties of properties, over properties of properties of properties, and so on. If any arbitrary, but finite, nesting is allowed, the resulting logic is often called *type theory*. One can also think about continuing this into the transfinite. All those logics are called higher-order logics (see Enderton 2015).

A specific incarnation of type theory is Church's type theory, which is often also called *simple type theory*. Alonzo Church formulated this type theory in Church (1940). As the name suggests, type theory assigns types to the entities (individuals, functions, etc.) the language speaks about. By doing so Russell's paradox can be avoided, since the paradoxical formulas become untypeable. This means, that it is impossible to assign a type to the set of all sets that do not contain themselves. Church's type theory utilizes Church's λ-calculus to denote functions. Furthermore, since properties, sets, and relations can be expressed as functions from entities to Booleans the notion of functions is primitive in Church's type theory (see Andrews 2014).

Today, a wide variety of variants and extensions of Chuch's type theory exist. The variant of logic used by the Leo-III prover is also called *extensional type theory with choice*. For an overview of the development of HOL in the context of automatization see Benzmüller and Miller (2014).

For the purpose of this thesis we only consider classical higher-order logic and classical quantified Boolean formulas and omit their intuitionistic variants. One central difference between classical and intuitionistic logic is that classical logic admits the law of the excluded middle $(A \vee \neg A)$ while intuitionistic logic does not. An overview of intuitionistic logic with a focus on first-order logic, the related philosophical questions and further references can be found in Moschovakis (2015).

2.1 The Syntax

We now introduce the version of Church's type theory which is used by the Leo-III prover. When using the abbreviation HOL, we refer to exactly this logic. We follow the syntax definition and the notations given by Steen, Wisniewski, and Benzmüller (2016).

The *types* of HOL are terms freely generated by the binary *type constructor* $\to$ and a finite set of base types T_0. This means that the base types in T_0 are types and if τ and ν are types, then $(\tau \to \nu)$ is a type too. The set T_0 contains at least the two *primitive types* o and ι. Intuitively the type o denotes Boolean values and ι is the domain of individuals. Furthermore, a type $(\tau \to \nu)$ is the type of

functions with domain type τ and codomain type ν. $\mathcal{T}$ is the set of all types and the letters τ and ν denote arbitrary types. More formally:

Definition 2.1.1 (Types and Primitive Types)**.** The set $\mathcal{T}_0 = \{o, \iota\}$ are the *primitive types*. Let $\mathcal{T}_i = \{\tau \to \nu \mid \tau, \nu \in \mathcal{T}_{i-1}\}$. Then

$$\mathcal{T} = \bigcup_{i \geq 0} \mathcal{T}_i$$

is the set of *types*.

To omit parenthesis, we assume that $\to$ associates to the right. Hence, $\alpha \to \beta \to \gamma$ denotes $(\alpha \to (\beta \to \gamma))$.

Now the *terms* of HOL can be defined. Let Σ be a *signature*: A countable infinite set of symbols, each annotated with a type. Those are the *constants*. If the constant c is in the signature and annotated with the type τ, we say that c is *of type* τ and write $c_\tau \in \Sigma$. All constants are *terms*. Furthermore, for each type τ we have a countable infinite set of variables $\mathcal{V}_\tau := \{x^1_\tau, x^2_\tau, \ldots\}$. All variables are terms too. From those primitive terms more complex terms can be constructed by using *abstraction* and *application*. Let S_ν be any term, then $(\lambda x^i_\tau.\, S_\nu)_{\tau \to \nu}$ is a term and if $S_{\tau \to \nu}$ and T_τ are terms, then $(S_{\tau \to \nu}\, T_\tau)_\nu$ is a term too. This construction can be written in a more compact form as a grammar:

Definition 2.1.2 (HOL Terms)**.** The *terms* of HOL are generated by the following grammar rule:

$$S, T ::= c_\tau \mid x^i_\tau \mid (\lambda x^i_\tau.\, S_\nu)_{\tau \to \nu} \mid (S_{\tau \to \nu}\, T_\tau)_\nu$$

Definition 2.1.3 (HOL Formulas)**.** The *formulas* of HOL are the terms of type o.

Readers familiar with first-order logic might notice a subtlety here: The distinction between atomic formulas (formulas without logical symbols) and regular formulas disappears. This allows us to define the usual logical connective as merely constants.

For HOL we require the set of constants Σ to contain a complete logical signature. The connectives for disjunction, negation, and, for each type, equality and universal quantification already form a complete logical signature. Hence, $\{\vee_{o \to o \to o}, \neg_{o \to o}\} \subset \Sigma$. Furthermore, Σ contains an equality constant $=^\tau_{\tau \to \tau \to o}$ and an universal quantification constant $\Pi^\tau_{(\tau \to o) \to o}$ for each type τ. Intuitively, $\Pi^\tau_{(\tau \to o) \to o}\, p_{\tau \to o}$ is true if the predicate $p_{\tau \to o}$ is true for every element in the type τ.

To declutter the syntax, we omit the type superscript. This results in the overloaded constants $=_{\tau\to\tau\to o}$ and $\Pi_{(\tau\to o)\to o}$ where τ is arbitrary. We will also omit the types if they are clear from the context. In this text we will use the letters S, T, and U as syntactic variables denoting arbitrary terms. If they are annotated with a type in the subscript (e.g. S_τ), they denote arbitrary terms of that type. Furthermore, we will use x_τ, y_τ, z_τ as alternative variable names to avoid the index superscript.

Variables can occur in a term either bound or free. The sets of free and bound variables can be defined by a simple recursion:

Definition 2.1.4 (Free Variables)**.** Given a HOL term S the *free variables* $\mathcal{F}_S$ of S are:

$$\begin{aligned} \emptyset &\text{ if } S = c_\tau \text{ and } c_\tau \in \Sigma \\ \{x_\tau^i\} &\text{ if } S = x_\tau^i \text{ and } x_\tau^i \in \mathcal{V}_\tau \\ \mathcal{F}_T - x_\tau^i &\text{ if } S = \lambda x_\tau^i.\, T \\ \mathcal{F}_T \cup \mathcal{F}_U &\text{ if } S = (T\, U)_\tau \end{aligned}$$

Definition 2.1.5 (Bound Variables)**.** Given a HOL term S the *bound variables* $\mathcal{B}_S$ of S are:

$$\begin{aligned} \emptyset &\text{ if } S = c_\tau \text{ and } c_\tau \in \Sigma \\ \emptyset &\text{ if } S = x_\tau^i \text{ and } x_\tau^i \in \mathcal{V}_\tau \\ \mathcal{B}_T \cup \{x_\tau^i\} &\text{ if } S = \lambda x_\tau^i.\, T \\ \mathcal{B}_T \cup \mathcal{B}_U &\text{ if } S = (T\, U)_\tau \end{aligned}$$

Note that variables can occur both free and bound in a term. For example, the variable x_o^1 occurs both free and bound in the formula $((\lambda x_o^1.\, x_o^1 \vee x_o^2)_{o\to o}\, x_o^1)_o$.

To avoid clustering the signature we define the other logical connectives as functions.

Definition 2.1.6 (Conjunction, Implication, and Disequality)**.** *Conjunction* is defined as $(\wedge)_{o\to o\to o} := \lambda x_o.\, \lambda y_o.\, \neg((\neg x_o) \vee (\neg y_o))$ and *implication* is defined as $(\to)_{o\to o\to o} := \lambda x_o.\, \lambda y_o.\, (\neg x_o) \vee y_o$. *Inequality* is defined for arbitrary types τ as $(\neq)_{\tau\to\tau\to o} := \lambda x_\tau.\, \lambda y_\tau.\, \neg(x_\tau = y_\tau)$.

We will write this operators in the usual infix notation. Let S_o and T_o be arbitrary formulas then we will write $S_o \wedge T_o$, $S_o \to T_o$, and $S_\tau \neq T_\tau$ for arbitrary terms S_τ, T_τ of arbitrary type τ.

Furthermore, the universal quantification can be used to define the usual quantifiers syntactically.

Definition 2.1.7 (For-All and Existential Quantifiers)**.**

$$\forall x_\tau.\, S_o := \Pi(\lambda x_\tau.\, S_o)$$
$$\exists x_\tau.\, S_o := \neg(\Pi(\lambda x_\tau.\, \neg(S_o)))$$

The connectives λ, $\forall$, and $\exists$ are the *binders*. To make the syntax clearer, we will allow lists of variables to appear in the head of the binders. Hence, $\lambda x_o, y_o.\, (x_o \wedge y_o)$ is the same as $\lambda x_o.\, \lambda y_o.\, (x_o \wedge y_o)$. Furthermore, parenthesis can be omitted as usual. Note that the logical operators and application are left associative. This interacts nicely with the right associativity of the type constructor. The term $(f_{\alpha\to\beta\to\gamma}\, a_\alpha\, b_\beta)_\gamma$ is equivalent to $((f_{\alpha\to(\beta\to\gamma)}\, a_\alpha)\, b_\beta)_\gamma$.

2.2 Two Semantics

Formulas by themselves are often not enough, one is generally also interested in their meaning. To this end we now give a short presentation of the semantic of Church's type theory.

This is not as straightforward as it is in the case of first-order logic. While the completeness of first-order logic was established in a seminal result by Gödel (1930). Incompleteness of second-order logic with standard semantic, however, is a direct consequence of Gödel's famous incompleteness theorem (Gödel 1931). To recover completeness a weaker notion of validity is needed. One such notion is the Henkin semantic (cf. Benzmüller and Miller 2014).

Both, standard and Henkin semantic, use similar concepts. We will introduce standard semantic as a stronger case of Henkin semantic. Again the definitions and notations given here follow Steen, Wisniewski, and Benzmüller (2016).

We start with the definition of a frame.

Definition 2.2.1 (Domains and Frames)**.** A *frame* is a collection $\{\mathcal{D}_\tau\}_{\tau\in\mathcal{T}}$ of non-empty sets, such that $\mathcal{D}_o = \{\mathbb{T}, \mathbb{F}\}$ and $\mathcal{D}_{\tau\to\nu} \subseteq \mathcal{D}_\nu^{\mathcal{D}_\tau}$ is a collection of functions from $\mathcal{D}_\tau$ to $\mathcal{D}_\nu$.

$\mathbb{T}$ and $\mathbb{F}$ are the entities representing truth and falsehood.

This definition does not put any restrictions on the domain of ι. If all the function domains contain all functions, then we call this frame a *standard frame*.

Definition 2.2.2 (Standard Frame)**.** A frame, where $\mathcal{D}_{\tau\to\nu} = \mathcal{D}_\nu^{\mathcal{D}_\tau}$ for all types τ and ν is called a *standard frame*.

As the name suggests, standard semantic only allows standard frames as possible interpretation, while Henkin semantic allow general frames. Before formalizing this notion we need to define the notion of interpretation and the valuation of a term.

Definition 2.2.3 (Interpretation)**.** An interpretation is a pair $\mathcal{M} = (\{\mathcal{D}_\tau\}_{\tau\in\mathcal{T}}, \mathcal{I})$ where $\{\mathcal{D}_\tau\}_{\tau\in\mathcal{T}}$ is a frame and $\mathcal{I}$ is a function from the set of constants Σ to $\bigcup_{\tau\in\mathcal{T}} \mathcal{D}_\tau$, such that $\mathcal{I}(c_\tau) \in \mathcal{D}_\tau$ for all $\tau \in \Sigma$.

A *variable assignment* is an arbitrary mapping $\sigma : \bigcup_{\tau\in\mathcal{T}} \mathcal{V}_\tau \to \bigcup_{\tau\in\mathcal{T}} \mathcal{D}_\tau$, such that for all $x \in \mathcal{V}_\tau$: $\sigma(x) \in \mathcal{D}_\tau$. We write $\sigma[z/x_\tau]$ for the variable assignment which maps x_τ to z and every other variable y_τ to $\sigma(y_\tau)$.

Definition 2.2.4 (Valuation)**.** Given an interpretation $\mathcal{M}$ and a variable assignment σ, the *valuation* $\|.\|^{\mathcal{M},\sigma}$ of a term is recursively defined as:

$$\begin{aligned}
\|c_\tau\|^{\mathcal{M},\sigma} &= \mathcal{I}(c_\tau) \text{ if } c_\tau \in \Sigma \\
\|x_\tau\|^{\mathcal{M},\sigma} &= \sigma(x_\tau) \text{ if } x_\tau \in \mathcal{V}_\tau \\
\|(S_{\tau\to\nu}\, T_\tau)\|^{\mathcal{M},\sigma} &= \|S_{\tau\to\nu}\|^{\mathcal{M},\sigma}(\|T_\tau\|^{\mathcal{M},\sigma}) \\
\|\lambda x_\tau.\, S_\nu\|^{\mathcal{M},\sigma} &= \left(f \,:\, z \mapsto \|S_\nu\|^{\mathcal{M},\sigma[z/x_\tau]}\right) \in \mathcal{D}_{\tau\to\nu}
\end{aligned}$$

From now on we assume that $\mathcal{I}$ assigns the usual denotations to the primitive logical connectives.

Definition 2.2.5 (Denotation of the Primitive Logical Connectives)**.** The interpretation of the primitive constants is:

- $\mathcal{I}(\vee_{o\to o\to o})$ is the function $(x, y) \mapsto \begin{cases} \mathbb{F}, & \text{if } x = \mathbb{F} \text{ and } y = \mathbb{F} \\ \mathbb{T}, & \text{otherwise.} \end{cases}$
- $\mathcal{I}(\neg_{o\to o})$ is the function which maps $\mathbb{T}$ to $\mathbb{F}$ and $\mathbb{F}$ to $\mathbb{T}$.
- $\mathcal{I}(=_{\tau\to\tau\to o})$ is the function $(x, y) \mapsto \begin{cases} \mathbb{T}, & \text{if } x = y \\ \mathbb{F}, & \text{otherwise.} \end{cases}$
- $\mathcal{I}(\Pi_{(\tau\to o)\to o})$ is the function $f \mapsto \begin{cases} \mathbb{T}, & \text{if for all } x \in \mathcal{D}_\tau : f(x) = \mathbb{T} \\ \mathbb{F}, & \text{otherwise.} \end{cases}$

Definition 2.2.6 (Standard Interpretation)**.** A *standard interpretation* is an interpretation $\mathcal{M} = (\{\mathcal{D}_\tau\}_{\tau\in\mathcal{T}}, \mathcal{I})$ where $\{\mathcal{D}_\tau\}_{\tau\in\mathcal{T}}$ is a standard frame.

Definition 2.2.7 (Henkin Interpretation)**.** A *Henkin interpretation* is an interpretation $\mathcal{M} = (\{\mathcal{D}_\tau\}_{\tau\in\mathcal{T}}, \mathcal{I})$ where $\{\mathcal{D}_\tau\}_{\tau\in\mathcal{T}}$ is a frame such that $\|.\|^{\mathcal{M},\sigma}$ is well defined (i.e. total) for all terms.

The totality of the valuation function has to be added as an additional constraint for Henkin interpretations, otherwise it is possible to have λ-terms whose corresponding function is not part of the domain.

A formula S_o is called standard (Henkin) valid, if and only if $\|s_o\|^{\mathcal{M},\sigma} = \mathbb{T}$ for every variable assignment σ and every standard (Henkin) interpretation $\mathcal{M}$. If for any given standard (Henkin) interpretation $\|s_o\|^{\mathcal{M},\sigma} = \mathbb{T}$, we call this interpretation a standard (Henkin) model. Otherwise it is called a standard (Henkin) counter model. Note that every standard model is also a Henkin model. We say a set of formulas $\{s_o^1, s_o^2, \ldots, s_o^n\}$ is valid, if and only if every formula s_o^i is valid.

As mentioned before, a consequence of Gödel's incompleteness theorem is the incompleteness of higher-order logic. This means, that there are *valid* formulas which are not provable in any consistent deduction system for higher-order logic. Henkin (1950) showed soundness and completeness for Henkin semantics[1] (cf. Benzmüller and Miller 2014). However, this comes at a cost. Higher-order logic with Henkin semantics is essentially many-sorted first-order logic (see Enderton 2015).

While HOL with Henkin semantics has essentially the same valid sentences as first-order logic, the convenient language is a justification for continued interest in it. Furthermore, Parikh (1973) has shown that there are exist theorems in arithmetic that have an arbitrary smaller proof in second-order logic then in first-order logic. In a similar spirit Boolos (1987) presented a theorem of first-order logic whose shortest proof is so large that it can not be conceived to be written down. The proof in second-order logic, on the other hand, fits comfortably onto a few pages.

While we give an overview on the deduction systems used in automatic theorem provers in the next section, more classical proof calculi might be of interest. A discussion of Frege-Hilbert style calculi can be found in Benzmüller and Miller (2014). Furthermore, it is worth mentioning, that the type expressions can be extended in various ways. One such way is to allow for free variables in type expressions, resulting in a polymorphic type theory. This is implemented in the Leo-III prover (see Steen, Wisniewski, and Benzmüller 2017).

1 Henkin himself called Henkin models *general models*.

2.3 Automating Higher-order Logic

Given some assumptions $s_o^1, s_o^2, \ldots, s_o^n$ and a hypothesis s_o, can we find a deduction in some proof calculus that shows s_o from the assumptions? If the question can be answered affirmatively, we say that s_o is provable from $s_o^1, s_o^2, \ldots, s_o^n$ and write $s_o^1, s_o^2, \ldots, s_o^n \vdash s_o$. The goal of an automatic theorem prover is to search for such a deduction without user input other than the problem. Unfortunately, HOL with Henkin semantic is only semi-decidable. Therefore, it is only possible to construct computer systems which can theoretically find a proof if one exists, but might not terminate if the goal is unprovable. In practice theorem proving systems may not be able to find proofs, even if they exist.

This section gives a rough overview of state-of-the-art automatic theorem proving. We start with an overview of currently used systems. Then subsection 2.3.1 will describe some aspects of the proof calculi commonly used in HOL provers. Subsection 2.3.2 describes the Leo-III prover in more detail. Finally, subsection 2.3.3 describes the TPTP infrastructure.

Automatic theorem proving is not the only way in which logical reasoning can be automatized. If a theorem is not provable, it might be possible to find a counter model. Nitpick by Blanchette and Nipkow (2010) is a tool which searches for finite counter models. Furthermore, *interactive theorem provers* construct proofs by interacting with the user. Examples of such systems are the Coq system (see Bertot and Castran 2010) and Isabelle/HOL (Nipkow, Paulson, and Wenzel 2002). The last system also incorporates Nitpick and the Sledgehammer (Paulson and Blanchette 2010) tool, which allows the user to use external, fully automated theorem provers for proof search.

The CADE ATP *System Competition* is an annual competition of automatic theorem proving systems. Its latest iteration was CASC-J8, which was part of The 8th International Joint Conference on Automated Reasoning in 2016[2]. In the higher-order theorems track four systems participated: Satallax (in two different configurations), LEO-II, Leo-III (in two different configurations), and Isabelle. While Leo-III is discussed in subsection 2.3.2, we will now give a short description of the other systems.

Leo-II is the successor of LEO-I by Benzmüller and Kohlhase (1998). The calculus used by LEO-II is called a RUE (resolution by unification and equality) calculus and equality with extensionality is supported natively. This removes the challenging extensionality axiom from the search space.

2 see: `http://www.cs.miami.edu/~tptp/CASC/J8/`

Unification is implemented as a variation of Huet's pre-unification. We discuss this algorithm in subsection 2.3.1. Furthermore, Leo-II supports cooperation with first-order provers (see Benzmüller, Paulson, et al. 2015).

Satallax uses a tableau calculus based approach in combination with a SAT solver. The reasoning process creates a series of potentially refutable formulas. Then propositional formulas corresponding to the meaning of those formulas are generated and handed to the SAT solver *mini*SAT. If the propositional formulas are unsatisfiable the original set of higher-order formulas is unsatisfiable (see Brown 2013).

Isabelle is an automatized version of the Isabelle/HOL system. While Isabelle/HOL is, as discussed above, foremostly an interactive system, it provides a wide set of automatic tools. While the individual tools are incomplete and will by themselves often fail to find a proof, a scheduler can be used to automatically try various configurations of the automatic tools[3].

This list of competitors from the CASC-J8 competition is of course not a exhaustive list of available HOL automation tools. Furthermore, it blatantly disregards the historical development of HOL automation. For more general overview see Benzmüller and Miller (2014).

2.3.1 Aspects of Proof Calculi for HOL

In this and the next section, we will describe some aspects of proof calculi used by automatic theorem provers. While multiple complete calculi exist and are used for automatic proof search, many aspects discussed here are shared between systems.

Lambda Terms

As outlined in section 2.1, HOL terms contain λ-expressions to syntactically represent functions. The proof calculus must be able to handle those expressions. The calculus of inference on λ-terms is the λ-calculus. For an introduction into the λ-calculus, and further references, see Alama (2016).

The first requirement to correctly handle λ-expressions is renaming of bound variables:

3 For a description of the automatic mode see: `http://www.cs.miami.edu/~tptp/CASC/J8/SystemDescriptions.html#Isabelle---2015`

$$\frac{\lambda x_\tau.\, S_\nu \qquad y_\tau \in \mathcal{V}_\tau \text{ and } y_\tau \notin \mathcal{F}_S \cup \mathcal{B}_S}{\lambda y_\tau.\, S_\nu[y_\tau/x_\tau]}\,\mathcal{R}(\alpha\text{-conv.})$$

The restrictions on the α-conversion rule ensure that the new variable y is fresh. This avoids erroneous renaming such as changing the term $\lambda x.\, \lambda y.\, x$ to $\lambda y.\, \lambda y.\, y$. Usually terms which only differ in the name of bound variables are identified with each other:

Definition 2.3.1 (α-equivalence)**.** A term s_τ is α-*equivalent* with a term t_τ, if t_τ can be constructed from s_τ by multiple applications of the α-conversion rule.

Explicit application of α-conversion can be omitted by using a nameless representation of the variables in λ-terms such as de-Bruijn indices (see Wisniewski, Steen, and Benzmüller 2015).

A substitution σ is a function from variables to terms, which differs from the identity function only for finitely many variables. We write t/x for the substitution which replaces the variable x with the term t. A substitution can be extended to a function from terms to terms by applying them to the free variables occurring in the input terms. When doing so, α-conversion must be applied to avoid capturing of free variables. E.g. when applying the substitution y/x to the term $\lambda y.\, x$ one must first rename the bound variable y to get the $\lambda z.\, y$. For a given substitution σ, we written $t[\sigma]$ for an α-equivalent term of the term t after applying the substitution σ.

Substitutions can be composed. Let σ, θ be two substitutions, then $t[\theta][\sigma]$ denotes the resulting term after applying first θ and then σ to t.

Secondly, the notion of applying an argument to a function needs to be expressed. This is done by the β-reduction rule.

$$\frac{(\lambda x_\tau.\, S_\nu)_{\tau\to\nu} T_\tau \qquad \mathcal{F}_T \cap (\mathcal{F}_S \cup \mathcal{B}_S) = \emptyset}{S_\nu[T_\tau/x_\tau]}\,\mathcal{R}(\beta\text{-red.})$$

As for substitutions the rule is restricted such that accidental binding of free variables is avoided. Therefore, it might be necessary to apply α-conversation before applying β-reduction.

The final rule is η-conversion, which expresses functional extensionality. Two functions are considered equivalent, if they map the all arguments to the same value. Formally, this is expressed by the following two rules:

$$\frac{\lambda x_\tau.\, f_{\tau\to\nu}\, x_\tau \qquad x_\tau \notin \mathcal{F}_f}{f_{\tau\to\nu}}\,\mathcal{R}(\eta\text{-conv.})$$

$$\frac{f_{\tau\to\nu} \qquad x_\tau \notin \mathcal{F}_f}{\lambda x_\tau.\, f_{\tau\to\nu}\, x_\tau}\, \mathcal{R}(\eta\text{-conv.}')$$

Again two terms, such that one term can be transformed into the other by multiple application of the two η-conversation rules are called η-equivalent.

Resolution

The fundamental idea behind the proof calculi used by Leo-II and Leo-III is *resolution*. While the Leo-III system extends and modifies this approach, as described in subsection 2.3.2, Leo-II is directly based on resolution.

Resolution was first introduced by Robinson (1965) for first-order logic and was subsequently lifted to higher-order logics. Andrews (1971) gave a proof of completeness for elementary type theory. Subsequently, Huet (1972) described a complete proof search method for elementary type theory. The approach taken by Leo-II is based on Resolution by Unification and Equality (rue-resolution), which again was first developed for first-order logic and subsequently lifted by Benzmüller (1999) to hol. A discussion of the evolution of higher-order resolution is presented in Benzmüller (2002) and Benzmüller (2015) gives a short description of the development and calculus used in Leo-II. The following presentation of resolution is based on the latter.

As a proof calculus, resolution proofs can only show that a contradiction can be derived from an input problem. To show the validity of conjecture it has to be negated first. The resolution rules operate on clauses and the proof is finished when the empty clause is found.

A clause is a disjunction of formulas. More exactly: A clause is a formula of the form $s_o^1 \vee s_o^2 \vee \cdots \vee s_o^n$. A set of the form $\{s_o^1, s_o^2, \ldots, s_o^n\}$ is often used instead of the explicit formula. This also removes the necessity to handle removal of repeated formulas and the commutativity of the conjunction explicitly. For a given clause C, we will write $C \vee t_o$ for $C \cup \{t_o\}$ and $C \vee D$ for $C \cup D$. From the definition of disjunction, we know that in any given interpretation and variable assignment a clause is true if at least one of its members is true. Hence, any valuation of the empty clause will always be false. A set *of clauses*, on the other hand, is a conjunction of clauses. Such a set is true, if and only if all of is member clauses are true. Therefore, the empty conjunction is always true.

Furthermore, we will add the *polarity* of a formula as an annotation: $[s_o]^\alpha$ for $\alpha \in \{tt, ff\}$. $[s = t]^{tt}$ is equivalent to $s = t$, $[s = t]^{ff}$ is equivalent to $s \neq t$, $[s]^{tt}$ is equivalent to s, and $[s]^{ff}$ is equivalent to $\neg s$.

The proof search procedure maintains a set of clauses and iteratively applies the calculus rules which will produce new clauses to add to the clause set. If the

empty clause is found, a proof for the unsatisfiability of the input problem has been found. Therefore, to prove the validity of $s_o^1, s_o^2, \ldots, s_o^n \vdash s_o$, the prover starts with the clause set $\{[s_o^1]^{tt}, [s_o^2]^{tt}, \ldots, [s_o^n]^{tt}, [s_o]^{ff}\}$.

The resolution rule is:

$$\frac{C \vee [s]^{tt} \qquad D \vee [t]^{ff}}{C \vee D \vee [s = t]^{ff}}\,\mathcal{R}(Res)$$

That is: If a clause of the form $C \vee [s]^{tt}$ and one of the form $D \vee [t]^{ff}$ is found in the clause set, the clause $C \vee D \vee [s = t]^{ff}$ can be added. The new term $[s = t]^{ff}$ is called the *unification constraint*. How the prover operates on unification constraints is described in the next section.

To understand the motivation for this rule it is helpful to sketch an argument for its soundness. To show soundness, it is necessary to argue, that if the preconditions are true (under a given interpretation and variable assignment) the conclusion is true too. If the preconditions are true, then either C or $[s]^{tt}$ and either D or $[t]^{ff}$ are true. If s and t are not equal, then the conclusion holds. On the other hand, if they are equal then either $[s]^{tt}$ or $[t]^{ff}$ is false, and therefore either C or D is true.

Note that the resolution rule together with unification is not a complete calculus. See Benzmüller (2015) for a presentation of the full calculus used in Leo-II.

Unification

As mentioned above, resolution introduces unification constraints which need to be handled by the prover. The inclusion of explicit unification constraints is a result of the pre-unification approach which is required to handle unification in hol.

Knight (1989) gives the following general definition for the unification problem:

> Given two terms of logic built up from function symbols, variables, and constants, is there a substitution of terms for variables that will make the two terms identical?

In the case of hol this definition means, that we are interested in replacing the free variables of hol terms with new hol terms, such that two given terms become equal. Such a substitution is called an unifier:

Definition 2.3.2 (Unifier)**.** Given two terms s and t, a substitution σ is a unifier of s and t if $s[\sigma] = t[\sigma]$.

In the case of first-order logic the existence of a unifier is always decidable and there is exactly one *most general unifier* (MGU). A unifier σ is a MGU of s and t, if for every other unifier θ there is a substitution τ such that $\tau(\sigma(x)) = \theta(x)$ for all variables x. In fact, there are linear time algorithms to find the most general unifier of two first-order terms, even thought they might not be the ones used in practice (see Knight 1989).

The following rule describes a possible application of unification during proof search:

$$\frac{\{u_o^1, u_o^2, \ldots, u_o^n\} \vee [s = t]^{ff} \qquad \sigma \text{ is the MGU of } s \text{ and } t}{\{u_o^1[\sigma], u_o^2[\sigma], \ldots, u_o^n[\sigma]\}}$$

Intuitively, this rule expresses the fact, that if we can find a substitution σ of the free variables in a clause, such that $s[\sigma] = t[\sigma]$, then the unification constraint $[s[\sigma] = t[\sigma]]^{ff}$ is certainly false and can be removed from the clause.

Unfortunately, in the case of higher-order logic finding a unifier is not decidable anymore and there might be infinitely many MGUs. Therefore, the rule we just described can not be used. To solve this, Huet (1975) presented the *pre-unification* algorithm to tightly incorporate higher-order unification into the proof search procedure. Unification problems are no longer solved eagerly, but first added as unification constraints to the clauses and then solved by appropriate calculus rules. LEO-II and Leo-III implement a variation of pre-unification extended by extensionality principles. For an overview of the unification rules used in LEO-II see again Benzmüller (2015).

While we know that unification for first-order terms is decidable, we might be interested in further fragments of HOL where unification is decidable. Such a fragment is the pattern fragment. Nipkow (1993) developed a complete unification algorithm for the pattern fragment and presented an implementation in a functional programming language. The definition of patterns used there is:

Definition 2.3.3 (Pattern)**.** A term t in β-normal form is a *(higher-order) pattern* if every free occurrence of a variable F is in a subterm $F\, u_1 \ldots u_n$ of t such that $u_1 \ldots u_n$ is η-equivalent to a list of distinct bound variables.

For example, the term $\lambda x.\, \lambda y.\, F\, x\, y$ is a pattern, while $\lambda x.\, F\, x\, x$ is a non-pattern (cf. Nipkow 1993).

Since first-order terms do not contain free function or predicate symbols they all are patterns and therefore pattern unification subsumes first-order unification. Pattern unification has been implemented in Leo-III and we use it to implement blocked clause elimination in section 4.4.

Primitive Substitution

The primitive substitution rule is used to guess instantiations for predicates in provers such as Leo-III. Similar rules were proposed by Huet (1972) as *splitting* and as primitive substitution by Andrews (1989).

Given a literal of the form $Q_{\tau\to o}\, S$ which appears in some clause of the HOL problem at hand, then primitive substitution will blindly guess the structure of Q from the operators $\neg$, $\vee$, Π, and $=$.

Hence, primitive substitution as realized by Leo-II and Leo-III is restricted to the outermost symbol. Only iterated application of the rule can reach deeper into the term structure.

During prove search, primitive substitution must be applied to ensure completeness and often plays the role of a fallback, when resolution is not applicable anymore. The classical example to illustrate the need for primitive substitution is the term $\exists x_o.\, x$ which is first processed to the singular unit clause $[x_o]^{ff}$. There is no resolution partner for this clause and the empty clause can not derived from this clause alone. Primitive substitution, however, might guess the structure of x_o to be $\neg y_o$ introducing the clause $[\neg y_o]^{ff}$ into the problem which normalizes to $[y_o]^{tt}$. Resolution easily derives the empty clause from $[x_o]^{ff}$ and $[y_o]^{tt}$ (cf. Benzmüller 2015).

Clausification

As mentioned above, resolution works on a set of clauses. Therefore, automatic theorem provers attempt to first transform the input problem such that it is a conjunction of disjunctions.

Problems in first-order logic and QBFs can be transformed into a conjunctive normal form where the only occurring logical connectives are in the topmost conjunction of disjunctions. For the case of QBFs this is discussed in section 3.1.

Clausification for HOL formulas is complicated by the free nesting of functions and operators. Logical connectives and even quantifiers can occur in arguments to functions and inside λ-terms. Nevertheless, various methods to aid the clausification of HOL terms exist. See Wisniewski, Steen, Kern, et al. (2016) for an overview.

When we say a HOL problem is in *clause form*, we mean that some processing has been done to the problem towards creating a conjunction of clauses.

Skolemization

Skolemization is part of the pre-processing step and is applied directly after clausification. Clausification will create a problem of the form

$$Q_1 x^1_{\tau_1}. Q_2 x^2_{\tau_2}. \ldots Q_i x^i_{\tau_i}. C_1 \wedge C_2 \wedge \cdots \wedge C_j$$

where the $Q_n \in \{\forall, \exists\}$ are quantifiers and the C_n are clauses.

Skolemization will replace the existentially quantified variables with function constants. That is, if the quantifiers contain $\exists x_i. \ldots$ each occurrence of x_i will be replaced by $(sk_i\, x_{\pi_1} \ldots x_{\pi_j})$ where sk is of appropriate type and the x_{π_i} are the variables where $Q_{\pi_i} = \forall$ and $\pi_i \leq i$. The freshly introduced functions sk_i are called *Skolem functions*.

Consider for example the term $\forall x_\tau. \exists y_\delta. P\, x\, y$, then skolemization will result in the term $\forall x_\tau. P\, x\, (sk_{\tau\to\delta}\, x)$.

In the case of HOL, skolemization requires either the inclusion of the axiom of choice, or it must be restricted in its application to be sound (cf. Benzmüller and Miller 2014). The automatic theorem provers Leo-II and Leo-III can use skolemization without problems, since they support choice implicitly.

We will encounter Skolem functions in subsection 3.3 to define the semantics of DQBFs and their translation to HOL.

2.3.2 The Leo-III Prover

Leo-III is the successor of the Leo-II theorem prover. Development started in 2014 and is mostly conducted in the Scala programming language. This is a departure from the OCaml language used to develop Leo-II. Besides improving upon the proof search by employing ordered para-modulation/superposition and agent based parallelism, Leo-III also supports some new features. One goal is to extend the type system by including support for type classes and type constructors similar to System F^ω. At the time of writing, type polymorphism was already implemented. Furthermore, embedding non-classical logics, such as modal logic, into HOL has been developed as a successful strategy to automate modal logic (see Benzmüller and Woltzenlogel Paleo 2015). Leo-III will feature native support for this (cf. Wisniewski, Steen, and Benzmüller 2014).

We will now describe three aspects of Leo-III, which are relevant for the preprocessing techniques presented in chapter 4. Since the preprocessing techniques are implemented as a part of the overall Leo-III system, we will discuss the software architecture and the agent based approach.

Leo-III is build upon the system platform *Leo*PARD (Wisniewski, Steen, and Benzmüller 2015), which provides data structures for term representation, in-

dexing and search. Furthermore, LEOPARD provides the general blackboard architecture used by the agent based parallelization. While LEOPARD was published independently of Leo-III, recent iterations of the platform have been developed as part of Leo-III. LEO-II, the predecessor of Leo-III, utilized a main loop which loops over a set of clauses while generating new clauses which do not change the satisfiability status of the problem and subsequently adds the new clauses to the set of clauses. In Leo-III, however, the reasoning tasks are separated into specialized agents. The agents autonomously decide when to execute their work. To find a proof, the agents utilize a blackboard onto which they can write, or delete content (e.g. clauses) from. Leo-III also provides interfaces to external systems, such as LEO-II, or the first order prover E (cf. Steen, Wisniewski, and Benzmüller 2016). Furthermore, as part of this thesis project an interface to the SAT solver *pico*SAT was developed (see subsection 5.1.1).

The calculus of Leo-III is – similar to classical resolution – refutation-based. Instead of trying to prove $s_o^1, s_o^2, \ldots, s_o^n \vdash x_o$ directly, x_o is negated and the prover tries to show that the set $\{s_o^1, s_o^2, \ldots, s_o^n, \neg x_o\}$ is inconsistent. Provers do this by adding new clauses until the empty clause is found. The para-modulation calculus is a refutation-based calculus, which is successfully applied by first-order provers. This approach extends naïve resolution with an appropriate handling of equality, and uses term orders to avoid redundancy in the generated clauses. Lifting this approach to HOL is complicated by the difficult nature of higher-order unification. For a sketch of para-modulation as it is applied in Leo-III and further references see Steen, Wisniewski, and Benzmüller (2016).

2.3.3 The TPTP Infrastructure

To test automatic theorem provers, one needs to collect a diverse library of HOL problem. The TPTP (Thousands of Problems for Theorem Provers) project provides such a library[4]. While the TPTP library was first designed for first order provers, it has been extended to HOL, HOL with polymorphic types, and various other logics (cf. Sutcliffe and Benzmüller 2010).

The TPTP language is used to represent input problems. The grammar of this language is designed such that it can be read directly as a Prolog term. Consider as an example the surjective Cantor's theorem:

$$\neg(\exists G_{\iota\to(\iota\to o)}.\, \forall F_{\iota\to o}.\, \exists x_\iota.\, (G\, x) = F)$$

4 The TPTP library is available at: `http://www.cs.miami.edu/~tptp/`

This theorem expresses the fact, that for some domains (i.e. infinite domains) there is no surjective function G from the set of individuals to set of sets of individuals. Note that in this case sets are expressed by functions of type $\iota \to o$ (predicates).

Listing 1 is the TPTP encoding of this problem. It can be found in the TPTP library under the name `SET557^1.p`[5]. Some comment lines have been omitted from the problem for the sake of brevity.

```
%------------------------------------------------------------
% File     : SET557^1 : TPTP v6.4.0. Released v3.6.0.
% Domain   : Set Theory
% Problem  : Cantor's theorem
% Version  : Especial.
% [...]
% Status   : Theorem
% Rating   : 0.29 v6.4.0, 0.33 v6.3.0, 0.40 v6.2.0, 0.43
↪ v5.5.0, 0.50 v5.4.0, 0.20 v5.3.0, 0.40 v5.2.0, 0.20 v4.1.0,
↪ 0.00 v4.0.1, 0.67 v4.0.0, 0.33 v3.7.0
% [...]
%------------------------------------------------------------
thf(surjectiveCantorThm,conjecture,(
    ~ ( ? [G: $i > $i > $o] :
        ! [F: $i > $o] :
        ? [X: $i] :
          ( ( G @ X )
          = F ) ) )).

%------------------------------------------------------------
```

Listing 1: Cantor's theorem as TPTP problem.

The `thf()` function is used to describe a HOL formula in the TPTP language. It takes as first argument a name followed by a role keyword. Available roles include `axiom`, `definition`, `hypothesis`, and `theorem` among others. Then the actual formula is given. Depending on the chosen logic, the syntax can differ significantly. In the case of HOL the universal quantifier is represented by `!` and a `?` represents the existential quantifier. Furthermore, function application

5 The problem can be found on the TPTP website too: `http://www.cs.miami.edu/~tptp/cgi-bin/SeeTPTP?Category=Problems&Domain=SET&File=SET557^1.p`

is made explicit as a result of the Prolog compatibility with the @ operator. For a complete overview of the language, see the annotated BNF grammar available on the TPTP web page[6].

All problems provided by the TPTP library are categorized into domains. The problem presented above is part of the *set theory* (SET) domain. Beside the categorization into domains, the TPTP project also provides tools to search for problems by various parameters such as the number of symbols, the status, or the rating of the problem. The rating of a problem is a number between 0.0 and 1.0 and represents the ratio of theorem provers that could solve the problem. A problem with rating 0.0 could be solved by all provers, whereas a problem with rating 1.0 is yet unsolved. The TPTP language is also widely used as a unified input language for theorem provers and also supports the representation of proofs. Furthermore, some additional projects are part of the TPTP infrastructure, such as the *SystemOnTPTP* service. This service provides a web interface for a collection of centrally hosted theorem proving systems (see Sutcliffe 2009).

6 http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html

3 Quantified Boolean Formulas

After introducing the quite expressive language of HOL, we will now introduce Quantified Boolean Formulas (QBF). One can define QBFs either as a fragment of HOL, or as a generalization of propositional formulas. We will start by giving a definition of the HOL fragment below and then define the generalization of propositional formulas in section 3.1.

The fragment of QBF formulas are formulas without equality in which only Boolean variables appear.

Definition 3.0.4 (The QBF Fragment)**.** A HOL formula t_o is in the QBF *fragment* if:

1. All variables in t_o are of type o,
2. Equality does not appear in t_o,
3. Lambda terms only appear in the form $\Pi(\lambda x^i_o.\, S_o)$, where S_o is a formula in the QBF fragment.

Because of the third restriction, lambda terms are only allowed to express universal ($\forall x_o.\, S$) and existential ($\exists x_o.\, S$) quantification.

The HOL formula $\forall x_o.\, \exists y_o.\, (\neg y \vee x) \wedge (\neg x \vee y)$ is a semantically valid formula in the QBF fragment, whereas $\exists y_o.\, \forall x_o.\, (\neg y \vee x) \wedge (\neg x \vee y)$ is counter satisfiable. Since the domain of Boolean values ($\mathbb{T}$ and $\mathbb{F}$) is finite and therefore only finitely many assignments to the variables in a formula from the QBF fragment exist, the validity of formulas in the QBF fragment is decidable.

In the next section we will define QBFs independently from HOL and we will introduce the notion of *solving* QBFs. Subsequently, section 3.2 describes some state of the art QBF solving systems. To do so we start with a list of systems which participated in a recent competitive evaluation of QBF solving systems. As we will see, the majority of those systems rely on preprocessing tools.

When generalizing from propositional formulas one does not need to stop at QBFs, but can go further to Dependently Quantified Boolean Formulas (DQBF). We will introduce DQBFs in section 3.3. While dependently quantified Boolean formulas do not correspond to a fragment of HOL as directly as vanilla QBFs do,

we will identify HOL formulas that correspond to DQBF problems. Some QBF preprocessing techniques have been lifted to DQBF solving. Those techniques are good candidates for lifting to HOL.

3.1 The Structure of Quantified Boolean Formulas

Quantified Boolean formulas (QBF) are generalizing propositional formulas by adding universal and existential quantifiers over truth values. We will now present QBFs without using the language of HOL. Our discussion is inspired by the discussion in Büning and Bubeck (2009). Let $\mathtt{V} = \{\mathtt{x}_1, \mathtt{x}_2, \dots\}$ be a countable infinite set of propositional variables, then the QBFs are:

Definition 3.1.1. Quantified Boolean Formulas

- Every $\mathtt{x}_i \in \mathtt{V}$ is a QBF,
- if $\mathtt{s}$ is a QBF, then so is $\neg\mathtt{s}$,
- if $\mathtt{s}$ and $\mathtt{t}$ are QBFs, then so is $\mathtt{s} \vee \mathtt{t}$,
- if $\mathtt{s}$ and $\mathtt{t}$ are QBFs, then so is $\mathtt{s} \wedge \mathtt{t}$,
- if $\mathtt{s}$ is a QBF and $\mathtt{x}_i \in \mathtt{V}$ is a variable, then $\exists\mathtt{x}_i.\,\mathtt{s}$ is a QBF,
- if $\mathtt{s}$ is a QBF and $\mathtt{x}_i \in \mathtt{V}$ is a variable, then $\forall\mathtt{x}_i.\,\mathtt{s}$ is a QBF.

The difference between pure propositional formulas and QBFs are the two last rules. If those two rules are not used, the resulting formulas are propositional formulas. The connectives $\vee$ and $\wedge$ express disjunction and conjunction and the quantifiers express quantification over truth values. Free ($\mathcal{F}_\mathtt{s}$) and bound ($\mathcal{B}_\mathtt{s}$) variables of a QBF $\mathtt{s}$, are defined in the same way as they are defined for HOL terms. Furthermore, we will use the same shorthand notations as for HOL formulas. The QBF that intuitively correspond the HOL formulas in the QBF fragment mentioned above are $\forall\mathtt{x}.\,\exists\mathtt{y}.\,(\neg\mathtt{y} \vee \mathtt{x}) \wedge (\neg\mathtt{x} \vee \mathtt{y})$ and $\exists\mathtt{y}.\,\forall\mathtt{x}.\,(\neg\mathtt{y} \wedge \mathtt{x}) \vee (\neg\mathtt{x} \wedge \mathtt{y})$.

To define the semantics of QBFs we again use the notion of an interpretation. A QBF-interpretation is a mapping $\mathtt{I} : \mathtt{V} \to \{\mathbb{T}, \mathbb{F}\}$. Given a QBF-interpretation $\mathtt{I}$, we can again extend this recursively to a mapping from QBFs to truth values.

Definition 3.1.2 (Valuation of Quantified Boolean Formulas)**.** Given a QBF-interpretation $\mathtt{I}$ then the valuation $\|.\|^\mathtt{I}$ is recursively defined on QBFs as:

$$\|\mathtt{x}\|^\mathtt{I} = \mathtt{I}(\mathtt{x}) \text{ if } \mathtt{x} \in \mathtt{V}$$

$$\|\neg \mathtt{s}\|^{\mathtt{I}} = \begin{cases} \mathbb{T}, & \text{if } \|\mathtt{s}\|^{\mathtt{I}} = \mathbb{F} \\ \mathbb{F}, & \text{otherwise.} \end{cases}$$

$$\|\mathtt{s} \vee \mathtt{t}\|^{\mathtt{I}} = \begin{cases} \mathbb{T}, & \text{if } \|\mathtt{s}\|^{\mathtt{I}} = \mathbb{T} \text{ or } \|\mathtt{t}\|^{\mathtt{I}} = \mathbb{T} \\ \mathbb{F}, & \text{otherwise.} \end{cases}$$

$$\|\mathtt{s} \wedge \mathtt{t}\|^{\mathtt{I}} = \begin{cases} \mathbb{T}, & \text{if } \|\mathtt{s}\|^{\mathtt{I}} = \mathbb{T} \text{ and } \|\mathtt{t}\|^{\mathtt{I}} = \mathbb{T} \\ \mathbb{F}, & \text{otherwise.} \end{cases}$$

$$\|\exists \mathtt{x}.\, \mathtt{s}\|^{\mathtt{I}} = \begin{cases} \mathbb{T}, & \text{if } \|\mathtt{s}\|^{\mathtt{I}[\mathbb{T}/x]} = \mathbb{T} \text{ or } \|\mathtt{s}\|^{\mathtt{I}[\mathbb{F}/x]} = \mathbb{T} \\ \mathbb{F}, & \text{otherwise.} \end{cases}$$

$$\|\forall \mathtt{x}.\, \mathtt{s}\|^{\mathtt{I}} = \begin{cases} \mathbb{T}, & \text{if } \|\mathtt{s}\|^{\mathtt{I}[\mathbb{T}/x]} = \mathbb{T} \text{ and } \|\mathtt{t}\|^{\mathtt{I}[\mathbb{F}/x]} = \mathbb{T} \\ \mathbb{F}, & \text{otherwise.} \end{cases}$$

Based on QBF-interpretations, validity and satisfiability of QBFs is defined in the same way as the for HOL formulas:

Definition 3.1.3 (Status of Quantified Boolean Formulas)**.** A QBF $\mathtt{s}$ is *valid*, if $\|\mathtt{s}\|^{\mathtt{I}} = \mathbb{T}$ for all QBF-interpretations $\mathtt{I}$. A QBF is *satisfiable*, if there is a QBF-interpretation $\mathtt{I}$, such that $\|\mathtt{s}\|^{\mathtt{I}} = \mathbb{T}$.

A QBF which is not valid is called *counter satisfiable* and a QBF which is not satisfiable is *unsatisfiable*.

If a QBF $\mathtt{s}$ is valid, then $\neg\mathtt{s}$ is unsatisfiable. Two QBFs $\mathtt{s}, \mathtt{t}$ are logically equivalent ($\mathtt{s} \equiv \mathtt{t}$) if $\|\mathtt{s}\|^{\mathtt{I}} = \|\mathtt{t}\|^{\mathtt{I}}$ for all QBF-interpretations $\mathtt{I}$.

Since the set of variables is countable infinite, there are infinitely many interpretations. However only those that differ on the free variables appearing in a formula really matter:

Lemma 3.1.1. Given a QBF $\mathtt{s}$ and two QBF-interpretations $\mathtt{I}_1$ and $\mathtt{I}_2$, such that $\mathtt{I}_1(\mathtt{x}) = \mathtt{I}_2(\mathtt{x})$ for all $\mathtt{x} \in \mathcal{F}_{\mathtt{s}}$, then $\|\mathtt{s}\|^{\mathtt{I}_1} = \|\mathtt{s}\|^{\mathtt{I}_2}$.

This lemma can easily be shown by a structural induction over the structure of QBFs. As consequence of this lemma, we know that the interpretation does not matter for closed QBFs (formulas without free variables) and that validity and satisfiability coincides in this case. Furthermore, a QBF $\mathtt{s}$ with free variables $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ is satisfiable if and only if the QBF $\exists \mathtt{x}_1.\, \exists \mathtt{x}_2.\, \ldots \exists \mathtt{x}_n.\, \mathtt{s}$ is valid.

Every QBF can be translated into a logically equivalent formula in *conjunctive normal form*. The description given here is loosely based on the description presented by Büning and Bubeck (2009). It is customary to first assume the

formula is in Negative Normal Form (NNF) . A formula is in negative normal form if negation only occurs immediately in front of variables. The NNF of any QBF can be constructed by using the following rules:

$$\neg\neg \mathtt{s} \equiv \mathtt{s}$$
$$\neg(\mathtt{s} \vee \mathtt{t}) \equiv \neg\mathtt{s} \wedge \neg\mathtt{t} \qquad \neg(\forall \mathtt{x}.\, \mathtt{s}) \equiv \exists \mathtt{x}.\, \neg\mathtt{s}$$
$$\neg(\mathtt{s} \wedge \mathtt{t}) \equiv \neg\mathtt{s} \vee \neg\mathtt{t} \qquad \neg(\exists \mathtt{x}.\, \mathtt{s}) \equiv \forall \mathtt{x}.\, \neg\mathtt{s}$$

After constructing the NNF, we construct the *prenex* form. A QBF $\mathtt{s}$ is in prenex form, if it has the form $Q_1\mathtt{x}_1.\, Q_2\mathtt{x}_2.\, \ldots Q_n\mathtt{x}_n.\, \mathtt{t}$ with $Q_i \in \{\forall, \exists\}$ where $\mathtt{t}$ is quantifier free and in NNF. We call $Q_1\mathtt{x}_1.\, Q_2\mathtt{x}_2.\, \ldots Q_n\mathtt{x}_n.$ the *prefix* and $\mathtt{t}$ the *matrix* of the formula in CNF. While the prenex form can easily be constructed by lifting the quantifiers from inside of conjunctions and disjunctions to the outside, one has to pay attention to variables being bound multiple times in different subterms. For example, the term $(\forall \mathtt{x}.\, \mathtt{s}) \vee (\forall \mathtt{x}.\, \mathtt{t})$ is not equivalent to $\forall \mathtt{x}.\, (\mathtt{s} \vee \mathtt{t})$ if $\mathtt{x}$ appears free both in $\mathtt{s}$ and $\mathtt{t}$. To solve this, one can rename bound variables first, such that they are never bound by more then one quantifier. This can easily be done and we will from now on assume that this has been done. Formulas that have this property are called *cleansed*. In practice, there are differed strategies to construct the prenex form which have measurable influence on the reasoning performance of QBF solvers. For an overview see Egly et al. (2004).

After the prenex form has been constructed, the formula can be transformed into a formula in conjunctive normal form (CNF). A formula is in CNF, if it is in prenex normal form and the matrix is a conjunction of clauses.

To naïvely construct the CNF from the NNF the distributive laws $\mathtt{s} \vee (\mathtt{t} \wedge \mathtt{u}) \equiv (\mathtt{s} \vee \mathtt{t}) \wedge (\mathtt{s} \vee \mathtt{u})$ and $\mathtt{s} \wedge (\mathtt{t} \vee \mathtt{u}) \equiv (\mathtt{s} \wedge \mathtt{t}) \vee (\mathtt{s} \wedge \mathtt{u})$ can be used. This method, however, can lead to a QBF in CNF which is exponentially longer then the original formula. To avoid this problem one can introduce new variables recursively as *names* for subterms, replacing subterms with their names and adding terms equating the now simplified terms with their new names. As a last step, the conjunction of the equivalency terms is formed. This method was first described by Tseitin 1983. Since new variables are introduced, the resulting formula is not logically equivalent to the original one, but is satisfiable if and only if the original formula is. Overall we get:

Lemma 3.1.2. Any quantified Boolean formula $\mathtt{s}$ can be transformed into a cleansed formula $\mathtt{t}$ in CNF which is satisfiable if and only if $\mathtt{s}$ is satisfiable. The time to construct $\mathtt{t}$ and the length of $\mathtt{t}$ is linear in the length of $\mathtt{s}$.

Since in formulas in CNF negation only appears directly in front of variables, we will call variables ($\mathtt{x}_i$) and negated variables ($\neg\mathtt{x}_i$) *literals*. If l is a literal,

then $\bar{l}$ is the negated literal i.e. if $l = \mathtt{x}_i$ then $\bar{l} = \neg\mathtt{x}_i$ and if $l = \neg\mathtt{x}_i$ then $\bar{l} = \mathtt{x}_i$. Obviously $\bar{\bar{l}} = l$. We will use the variables $\mathtt{x}_1, \mathtt{x}_2, \ldots$ for variables bound by the universal quantifiers and $\mathtt{y}_1, \mathtt{y}_2, \ldots$ for existentially quantified variables. We write $\mathrm{var}(l)$ for the variable of a literal. I.e. $\mathrm{var}(\mathtt{x}) = \mathtt{x}$ and $\mathrm{var}(\neg\mathtt{x}) = \mathtt{x}$. Furthermore, a literal l is a $\forall$-literal, if $\mathrm{var}(l)$ is bound by a universal quantifier. On the other hand, a $\exists$-literal l is a literal with $\mathrm{var}(l)$ bound by an existential quantifier. We express the *polarity* of a literal as $\mathrm{pol}(l)$. That means, that $\mathrm{pol}(\mathtt{x}) = \mathbb{T}$ and $\mathrm{pol}(\neg\mathtt{x}) = \mathbb{F}$.

Deciding satisfiability of QBFs is solvable in polynomial space and PSPACE-complete. There is a close correspondence between QBFs and the polynomial-time hierarchy. The complexity depends on the number of quantifier alternations in the prefix.

Definition 3.1.4 (Prefix Type of Quantified Boolean Formulas)**.** A QBF which is propositional has prefix type $\Sigma_0 = \Pi_0$ (empty prefix). If a formula $\mathtt{s}$ has prefix type Σ_n, then the formula $\forall\mathtt{x}_1. \ldots \forall\mathtt{x}_m. \mathtt{s}$ is of type Π_{n+1}. If a formula $\mathtt{s}$ has prefix type Π_n, then the formula $\exists\mathtt{y}_1. \ldots \exists\mathtt{y}_m. \mathtt{s}$ is of type Σ_{n+1}.

The correspondence between QBFs and the polynomial-time hirachy is:

Theorem 3.1.3. For $k \geq 1$, the satisfiability problem for QBFs with prefix type Σ_k is Σ_k^P-complete, and for formulas with prefix type Π_k, it is Π_k^P-complete (cf. Büning and Bubeck 2009).

The class Σ_k^P is the class of all problems which can be decided in polynomial time by a non-deterministic Turing machine with the help of a Σ_{k-1}^P-oracle. A $\mathcal{C}$-oracle is a subroutine which allows the program to solve a problem in the complexity class $\mathcal{C}$ in constant time. The class Π_k^P contains every problem whose complement is in Σ_k^P. Furthermore, $\Sigma_1^P = NP$ and $\Pi_1^P = coNP$.

While we only consider classical QBF in this work, intuitionistic QBF has some applications too. In fact, it corresponds to the type system *System F* [1]. Since the type inhibition problem for System F is undecidable, this also means that the satisfiability of intuitionistic QBFs is undecidable.

Resolution for Quantified Boolean Formulas

Resolution for QBFs was first introduced by Büning, Karpinski, and Flogel (1995). In contemporary literature resolution for QBFs is often called Q-resolution. This emphasizes the explicit handling of universally quantified variables.

1 See this blog post by Aaron Stump: `https://queuea9.wordpress.com/2011/03/24/quantified-boolean-conundrum/`.

Alternatively, it is possible to use propositional resolution together with instantiation of the universally quantified variables. See Janota and Marques-Silva (2015) for a proof theoretic comparison.

As in the case of RUE-resolution, Q-resolution attempts to show the unsatisfiability of a problem by deriving the empty clause. To show the validity of an arbitrary QBF the formula is first negated and then transformed into CNF.

The Q-resolution rule is:

$$\frac{C \vee l \qquad D \vee \bar{l} \qquad l \text{ is an } \exists\text{-literal}}{\underbrace{(\mathcal{R}_\forall(C) \vee \mathcal{R}_\forall(D)) \setminus \{l, \bar{l}\}}_{\text{not tautological}}}\ \mathcal{R}(\text{Q-Res})$$

As indicated by the first rule, Q-resolution works by removing complementary $\exists$-literals as long as the resulting clause is not tautological. A clause is a tautological if it cointains two complementary literals. Furthermore, Q-resolution performs *universal reduction* as indicated by $\mathcal{R}_\forall(.)$. Given a clause C, $\mathcal{R}_\forall(C)$ is the clause after removing all $\forall$-literals which are not before any $\exists$-literal in the clause. A literal l_1 is said to be before a literal l_2, if the variable of l_1 occurs in the prefix left of the variable of l_2. If l_1 is before l_2, we will write $l_1 < l_2$.

In other words, universal reduction removes all universally quantified variables which no existentially quantified variables depend on. Since a clause consisting only of non-complementary $\forall$-literals is always false, deriving such a clause is enough to finish the proof. If the resulting clause, however, would be tautological, the Q-resolution rule is not applicable. Not only are tautological clauses useless to finish the proof, they are harmful in combination with universal reduction[2].

Consider the formula

$$\exists \mathtt{y}_1.\, \forall \mathtt{x}.\, \exists \mathtt{y}_2.\, \mathtt{y}_1 \wedge (\neg \mathtt{y}_1 \vee \mathtt{x} \vee \mathtt{y}_2) \wedge (\neg \mathtt{x} \vee \neg \mathtt{y}_2).$$

This formula is valid and therefore Q-resolution should not be able to derive the empty clause. Consider, however, the following derivation where a disallowed tautology is generated:

$$\dfrac{\dfrac{\mathtt{y}_1 \qquad \dfrac{\neg \mathtt{y}_1 \vee \mathtt{x} \vee \mathtt{y}_2 \qquad \neg \mathtt{x} \vee \neg \mathtt{y}_2}{(\mathcal{R}_\forall(\neg \mathtt{y}_1 \vee \mathtt{x} \vee \mathtt{y}_2) \vee \mathcal{R}_\forall(\neg \mathtt{x} \vee \neg \mathtt{y}_2)) \setminus \{\mathtt{y}_2, \neg \mathtt{y}_2\}}\ \mathcal{R}(\text{Q-Res}')}{(\mathcal{R}_\forall(\mathtt{y}_1) \vee \mathcal{R}_\forall(\neg \mathtt{y}_1 \vee \mathtt{x} \vee \neg \mathtt{x})) \setminus \{\mathtt{y}_1, \neg \mathtt{y}_1\}}\ \mathcal{R}(\text{Q-Res})}{\{\}}$$

2 Martin Suda pointed out this subtlety in a personal conversation. As he noticed, first-order (and higher-order) resolution does allow tautologies. However in those cases unification does block the removal of tautologies.

The first resolution step generates the tautology $\neg \mathtt{y}_1 \vee \mathtt{x} \vee \neg \mathtt{x}$. Universal reduction is unable to delete the variable $\mathtt{x}$ from this clause, because $\mathtt{y}_2$ is still dependent on it. In the second resolution step, with $\mathtt{y}_1$, the tautology is removed by universal reduction which is obviously unsound.

While universal reduction can be applied as a standard simplification rule, it is also necessary for the completeness of Q-resolution (cf. Büning and Bubeck 2009).

From Quantified Boolean Formulas to Higher Order Logic

We will now show that QBFs correspond to the QBF fragment of HOL. The translation between the two formalisms is straightforward.

Definition 3.1.5 (Translation from QBF to HOL)**.** The function ρ from QBF formulas to HOL formulas is defined recursively:

$$\begin{aligned} \rho(\mathtt{x}_i) &= x_o^i & \rho(\neg \mathtt{s}) &= \neg_{o\to o} \rho(\mathtt{s}) \\ \rho(\mathtt{s} \vee \mathtt{t}) &= \rho(\mathtt{s}) \vee_{o\to o\to o} \rho(\mathtt{t}) & \rho(\mathtt{s} \wedge \mathtt{t}) &= \rho(\mathtt{s}) \wedge_{o\to o\to o} \rho(\mathtt{t}) \\ \rho(\exists \mathtt{x}_i.\, \mathtt{s}) &= \exists x_o^i.\, \rho(\mathtt{s}) & \rho(\forall \mathtt{x}_i.\, \mathtt{s}) &= \forall x_o^i.\, \rho(\mathtt{s}) \end{aligned}$$

By definition 3.0.4, all formulas generated by ρ are in the QBF fragment of HOL. Furthermore, the function is invertible and the inverse is defined on all formulas of the QBF fragment. Therefore, this function is a bijection between QBFs and the QBF fragment of HOL.

The satisfiability of the two formalisms corresponds:

Theorem 3.1.4. Let $\mathtt{s}$ be an arbitrary QBF, then there is an interpretation $\mathcal{M}$ and variable assignment σ such that $\|\rho(\mathtt{s})\|^{\mathcal{M},\sigma} = S$ if and only if there is a QBF-interpretation $\mathtt{I}$, such that $\|s\|^{\mathtt{I}} = S$ where $S \in \{\mathbb{T}, \mathbb{F}\}$.

Proof. Given any QBF-interpretation $\mathtt{I}$, let $\sigma_{\mathtt{I}} : x_o^i \mapsto \|\mathtt{x}_i\|^{\mathtt{I}}$ be the corresponding variable assignment . Since no formula in the QBF fragment contains variables which are not of type o, this assignment is well defined for all formulas in the QBF fragment. Furthermore, we can choose an arbitrary HOL interpretation, since the domain will always contain both truth values and the functions for the logical operators.

Given an interpretation $\mathcal{M}$ and a variable assignment σ, then the corresponding QBF-interpretation $\mathtt{I}_\sigma$ is defined as: $\mathtt{I}_\sigma : \mathtt{x}_i \mapsto \sigma(x_o^i)$.

Now first assume a QBF $\mathtt{s}$ and a QBF-interpretation $\mathtt{I}$, such that $\|\mathtt{s}\|^{\mathtt{I}} = S$. We have to show that $\|\rho(\mathtt{s})\|^{\mathcal{M},\sigma_{\mathtt{I}}} = S$. This can be shown by induction over the structure of QBFs.

As the base case for the induction we have to consider terms which are just variables. By definition the corresponding interpretation will assign the same value to the mapped HOL variable in this case.

Now we do a case distinction on the topmost operator:

- If $\mathtt{s} = \neg\mathtt{t}$ and
  1. $\|\neg\mathtt{t}\|^{\mathtt{I}} = \mathbb{T}$, then $\|\mathtt{t}\|^{\mathtt{I}} = \mathbb{F}$ and by applying the induction hypothesis $\|\rho(\neg\mathtt{t})\|^{\mathcal{M},\sigma_{\mathtt{I}}} = \|\neg\rho(\mathtt{t})\|^{\mathcal{M},\sigma_{\mathtt{I}}} = \mathcal{I}(\neg)(\|\rho(\mathtt{t})\|^{\mathcal{M},\sigma_{\mathtt{I}}}) = \mathcal{I}(\neg)(\mathbb{F}) = \mathbb{T}$.
  2. $\|\neg\mathtt{t}\|^{\mathtt{I}} = \mathbb{F}$, then $\|\mathtt{t}\|^{\mathtt{I}} = \mathbb{T}$ and $\|\rho(\neg\mathtt{t})\|^{\mathcal{M},\sigma_{\mathtt{I}}} = \mathcal{I}(\neg)(\mathbb{T}) = \mathbb{F}$
- If $\mathtt{s} = \mathtt{t} \vee \mathtt{u}$ and
  1. $\|\mathtt{t}\vee\mathtt{u}\|^{\mathtt{I}} = \mathbb{T}$, then without loss of generality $\|\mathtt{t}\|^{\mathtt{I}} = \mathbb{T}$ and by induction hypothesis $\|\rho(\mathtt{t} \vee \mathtt{u})\|^{\mathcal{M},\sigma_{\mathtt{I}}} = \mathcal{I}(\vee)(\|\rho(\mathtt{t})\|^{\mathcal{M},\sigma_{\mathtt{I}}}, \|\rho(\mathtt{u})\|^{\mathcal{M},\sigma_{\mathtt{I}}}) = \mathcal{I}(\vee)(\mathbb{T}, \_) = \mathbb{T}$.
  2. $\|\mathtt{t} \vee \mathtt{u}\|^{\mathtt{I}} = \mathbb{F}$, then $\|\mathtt{t}\|^{\mathtt{I}} = \mathbb{F}$ and $\|\mathtt{u}\|^{\mathtt{I}} = \mathbb{F}$. We get by induction hypothesis $\|\rho(\mathtt{t} \vee \mathtt{u})\|^{\mathcal{M},\sigma_{\mathtt{I}}} = \mathcal{I}(\vee)(\|\rho(\mathtt{t})\|^{\mathcal{M},\sigma_{\mathtt{I}}}, \|\rho(\mathtt{u})\|^{\mathcal{M},\sigma_{\mathtt{I}}}) = \mathcal{I}(\vee)(\mathbb{F}, \mathbb{F}) = \mathbb{F}$.
- If $\mathtt{s} = \mathtt{t} \wedge \mathtt{u}$ and
  1. $\|\mathtt{t} \wedge \mathtt{u}\|^{\mathtt{I}} = \mathbb{T}$, then $\|\mathtt{t}\|^{\mathtt{I}} = \mathbb{T}$ and $\|\mathtt{u}\|^{\mathtt{I}} = \mathbb{T}$. We get by induction hypothesis $\|\rho(\mathtt{t} \wedge \mathtt{u})\|^{\mathcal{M},\sigma_{\mathtt{I}}} = \mathcal{I}(\wedge)(\|\rho(\mathtt{t})\|^{\mathcal{M},\sigma_{\mathtt{I}}}, \|\rho(\mathtt{u})\|^{\mathcal{M},\sigma_{\mathtt{I}}}) = \mathcal{I}(\wedge)(\mathbb{T}, \mathbb{T}) = \mathbb{T}$.
  2. $\|\mathtt{t}\wedge\mathtt{u}\|^{\mathtt{I}} = \mathbb{F}$, then without loss of generality $\|\mathtt{t}\|^{\mathtt{I}} = \mathbb{F}$ and by induction hypothesis $\|\rho(\mathtt{t} \wedge \mathtt{u})\|^{\mathcal{M},\sigma_{\mathtt{I}}} = \mathcal{I}(\wedge)(\|\rho(\mathtt{t})\|^{\mathcal{M},\sigma_{\mathtt{I}}}, \|\rho(\mathtt{u})\|^{\mathcal{M},\sigma_{\mathtt{I}}}) = \mathcal{I}(\wedge)(\mathbb{F}, \_) = \mathbb{F}$.
- If $\mathtt{s} = \exists\mathtt{x}_i.\,\mathtt{s}$ and
  1. $\exists\mathtt{x}_i.\,\mathtt{s} = \mathbb{T}$: Then there is some $S \in \{\mathbb{T}, \mathbb{F}\}$, such that $\|\mathtt{s}\|^{\mathtt{I}[S/x]} = \mathbb{T}$ and we get $\|\rho(\mathtt{s})\|^{\mathcal{M},\sigma_{\mathtt{I}[S/x]}} = \mathbb{T} = \|\rho(\exists\mathtt{x}_i.\,\mathtt{s})\|^{\mathcal{M},\sigma_{\mathtt{I}}}$.
  2. $\exists\mathtt{x}_i.\,\mathtt{s} = \mathbb{F}$: Then $\|\rho(\mathtt{s})\|^{\mathcal{M},\sigma_{\mathtt{I}[S/x]}} = \|\mathtt{s}\|^{\mathtt{I}[S/x]} = \mathbb{F}$ for both $S \in \{\mathbb{T}, \mathbb{F}\}$ and therefore $\|\rho(\exists\mathtt{x}_i.\,\mathtt{s})\|^{\mathcal{M},\sigma_{\mathtt{I}}} = \mathbb{F}$.
- If $\mathtt{s} = \forall\mathtt{x}_i.\,\mathtt{s}$ and
  1. $\forall\mathtt{x}_i.\,\mathtt{s} = \mathbb{T}$: Then $\|\rho(\mathtt{s})\|^{\mathcal{M},\sigma_{\mathtt{I}[S/x]}} = \|\mathtt{s}\|^{\mathtt{I}[S/x]} = \mathbb{T}$ for both $S \in \{\mathbb{T}, \mathbb{F}\}$ and therefore $\|\rho(\forall\mathtt{x}_i.\,\mathtt{s})\|^{\mathcal{M},\sigma_{\mathtt{I}}} = \mathbb{T}$.
  2. $\forall\mathtt{x}_i.\,\mathtt{s} = \mathbb{F}$: Then there is some $S \in \{\mathbb{T}, \mathbb{F}\}$, such that $\|\mathtt{s}\|^{\mathtt{I}[S/x]} = \mathbb{F}$ and we get $\|\rho(\mathtt{s})\|^{\mathcal{M},\sigma_{\mathtt{I}[S/x]}} = \mathbb{F} = \|\rho(\forall\mathtt{x}_i.\,\mathtt{s})\|^{\mathcal{M},\sigma_{\mathtt{I}}}$.

Constructing the QBF-interpretation from a given HOL interpretation and variable assignment is similar. □

Theorem 3.1.4 covers satisfiability and counter satisfiability. Therefore, we get as an immediate result:

Corollary 3.1.5. Let $\mathtt{s}$ be an arbitrary QBF, then $\rho(\mathtt{s})$ is *satisfiable* if and only if $\mathtt{s}$ is.

Since validity of a formula is equivalent to the unsatisfiability of its negation and for all quantified Boolean formulas $\rho(\neg\mathtt{s}) = \neg_{o\to o}\rho(\mathtt{s})$, theorem 3.1.4 also shows:

Corollary 3.1.6. Let $\mathtt{s}$ be an arbitrary QBF, then $\rho(\mathtt{s})$ is *valid* if and only if $\mathtt{s}$ is.

3.2 The Solving Pipeline

After introducing QBFs in the last section, we will have a look at state of the art QBF solving. What the CASC is for HOL provers, is QBFEVAL for QBF solvers. The latest iteration of this competitive evaluation, QBFEVAL'16, was part of the SAT 2016 conference in Bordeaux, France. See the report by Pulina (2016) and the web page of QBFEVAL[3] for an extensive description of the competition. The competition featured multiple tracks. Amongst them are tracks for QBFs in prenex CNF, for QBFs in prenex non-CNF form, and for proof producing solvers.

Overall 22 systems participated in the evaluation, some of them in multiple configurations for a total of 44 systems. The prenex CNF track featured 12 systems in overall 19 configurations[4]. Of those 12 systems all, but the two systems which do not reason on prenex CNF formulas (AIG*solve* and G*host*textscq), integrate a preprocessing tool in at least one configuration. Therefore, we will first describe the three systems which solved the most problems in the prenex CNF track and then discuses the used preprocessing tools.

RAReQS by Janota, Klieber, et al. (2012) is based on Counterexample Abstraction Refinement (CEGAR). The solver step by step expands the formula into a propositional formula by replacing variables with Boolean

3 `http://www.qbflib.org/index_eval.php`

4 The prenex CNF track originally had 24 entrants. Some systems, however, have been discarded from the results. See Pulina (2016) for details.

constants. For existentially quantified variables a disjunction, for universally quantified variables a conjunction is introduced. The resulting propositional formula can then be solved by a SAT solver. To mediate the exponential growth of the formulas generated by this procedure, the system expands only a selected subset of the variables. This subset is called an *abstraction*. The solver then uses solutions for the propositional formulas induced by a given abstraction as candidate solutions. Candidate solutions are not necessary solutions for the original problem, but the solver uses counterexamples for candidate solutions to refine the abstraction. The version of RAR*e*QS submitted to QBFEVAL'16 uses the preprocessing tool BLOQQER. RAR*e*QS solved 640 of 825 problems.

QSTS is based on nested SAT solving. The solver first applies a tool to transform the input problem into an internal SAT-TO-SAT format while applying various transformations and a structure extraction algorithm. Then the SAT-TO-SAT solver is called. This tool is a general purpose SAT solver that extends classical propositional satisfiability with nesting of SAT solving (see Bogaerts, Janhunen, and Tasharrofi 2016). The most successful configuration of QSTS used BLOQQER as a preprocessor and solved 613 of 825 problems.

DepQBF by Lonsing and Biere (2010) is a search based QBF solver. The search procedure guesses variable assignments and then propagates assignments necessary implied by the guessed assignment. If a contradiction is found, the procedure backtracks the implication process and tries another assignment. A number of techniques are used to speed this process up, most notably the addition of clauses found contradictory to exclude new, but ultimately unsatisfying assignments early on. Overall the algorithm implemented by D*ep*QBF is QDPLL with conflict-driven clause and solution-driven cube learning. As the name suggests, it is an extension of the successful DPLL approach.

While some configurations of D*ep*QBF, which participated in QBFEVAL'16 use the BLOQQER preprocessing tool, version 5.0 of D*ep*QBF also supports the use of the preprocessing technique blocked clause elimination during reasoning (see Lonsing, Bacchus, et al. 2015). The most successful configuration of D*ep*QBF solved 603 of 825 problems.

While the top three QBF solvers all use quite different solving techniques, they all used the preprocessing tool BLOQQER. This indicates that competitive QBF solving relies on a pipeline approach.

First, the problem is processed by one, or multiple preprocessing tools. The preprocessing tools usually accept the problem in CNF prenex form using a standardized input format. The format used for the competition is the QDIMACS format. A description of the QDIMACS format can be found on page 73. The output of the preprocessing tool is a satisfiability-equivalent problem which again is in the QDIMACS format. Secondly, the preprocessed file is consumed by the solver proper which outputs the status of the problem.

While the preprocessing tool BLOQQER is widely used, it is not the only preprocessing tool used by systems which participated in QBFEVAL'16. The HIQQER system used the two preprocessors PLODDER and EQXBF (cf. Janota, Jordan, et al. 2016). In one of its configurations the QSTS tool uses BREAKID (Devriendt, Bogaerts, and Bruynooghe 2014), a symmetry breaking tool for propositional problems which has been extended to QBF. This particular configuration, however, has been omitted from the result because it reported discrepancies. Finally, the preprocessing tool *sQueeze*BF (Giunchiglia, Marin, and Narizzano 2010) has been coupled with the A*qua*, AQME, and S*tru*QS system.

The BREAKID tool is developed publicly[5] and BLOQQER is available on its web page[6]. We could not find publicly available versions of the other mentioned preprocessing tools.

As the name BLOQQER indicates, the main preprocessing technique it uses is Quantified Blocked Clause Elimination (QBCE) (Biere, Lonsing, and Seidl 2011). It also implements a wide variety of other preprocessing techniques and recent versions can produce a proof output in the QRAT format (Heule, Seidl, and Biere 2014). We will give a general overview on some preprocessing techniques for QBFs in subsection 4.1.

3.3 Dependently Quantified Boolean Formulas

Dependently Quantified Boolean Formulas (DQBF) are a generalization of QBF which does not form a fragment of HOL. DQBFs, however, can be easily translated to satisfiability equivalent HOL formulas. Preprocessing techniques which have been generalized from QBF solving to DQBF solving are good candidates for lifting to HOL.

As we discussed in the previous sections, the variables in the quantifier prefix of a QBF in prefix normal form are linearly ordered. One can drop this restriction and extend the syntax with a way to explicitly declare the dependencies of the existentially quantified variables from the universally quantified variables.

5 at: `https://bitbucket.org/krr/breakid`
6 at: `http://fmv.jku.at/bloqqer/`

This extension of QBF was first defined by Peterson and Reif (1979) to model games with partially hidden information. Their definition of DQBFs is restricted to formulas in prenex normal form and while it is straightforward to define the syntax of arbitrary DQBFs, we follow their definition:

Definition 3.3.1 (Dependently Quantified Boolean Formulas)**.** The DQBFs are formulas of the form

$$\forall \mathtt{x}_1. \forall \mathtt{x}_2. \ldots \forall \mathtt{x}_n. \exists \mathtt{y}_1(\mathtt{x}_{l_1^1}, \mathtt{x}_{l_1^2}, \ldots, \mathtt{x}_{l_1^{n_1}}). \ldots \exists \mathtt{y}_m(\mathtt{x}_{l_m^1}, \ldots, \mathtt{x}_{l_m^{n_m}}). \Phi$$

where $1 \leq l_i^j \leq n$ and Φ is a propositional formula with the free variables $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n, \mathtt{y}_1, \mathtt{y}_2, \ldots, \mathtt{y}_m$.

In this new formulation, the existentially quantified variables carry the list of universally quantified variables with them. This obviously makes multiple quantifier alternations superfluous. Furthermore, the order and multiplicity of the variables denoting the dependencies does not matter. Therefore, for quantifiers of the form $\exists \mathtt{y}_i(\mathtt{x}_{l_i^1}, \mathtt{x}_{l_i^2}, \ldots, \mathtt{x}_{l_i^{n_i}}). \ldots$ we can collect universally quantified variables in a set $D_{\mathtt{y}_i} = \{\mathtt{x}_{l_i^1}, \mathtt{x}_{l_i^2}, \ldots, \mathtt{x}_{l_i^{n_i}}\}$. The set $D_{\mathtt{y}_i}$ is the *dependency set* of the existentially quantified variable $\mathtt{y}_i$.

Every QBF can be expressed as a DQBF by collecting all universally quantified variables that appear before an existentially quantified variable $\mathtt{y}_i$ in the prefix into the dependency set $D_{\mathtt{y}_i}$. So $\forall \mathtt{x}. \exists \mathtt{y}. (\neg \mathtt{y} \vee \mathtt{x}) \wedge (\neg \mathtt{x} \vee \mathtt{y})$ corresponds to $\forall \mathtt{x}. \exists \mathtt{y}(\mathtt{x}). (\neg \mathtt{y} \vee \mathtt{x}) \wedge (\neg \mathtt{x} \vee \mathtt{y})$ and if we switch the quantifiers to $\exists \mathtt{y}. \forall \mathtt{x}. (\neg \mathtt{y} \vee \mathtt{x}) \wedge (\neg \mathtt{x} \vee \mathtt{y})$ the corresponding DQBF changes to $\forall \mathtt{x}. \exists \mathtt{y}(\,). (\neg \mathtt{y} \vee \mathtt{x}) \wedge (\neg \mathtt{x} \vee \mathtt{y})$. Not all DQBFs correspond to QBFs. Consider for example the formula

$$\forall \mathtt{x}_1, \mathtt{x}_2. \exists \mathtt{y}_1(\mathtt{x}_1). \exists \mathtt{y}_2(\mathtt{x}_2). (\mathtt{y}_1 \vee \mathtt{x}_1) \wedge (\neg \mathtt{y}_1 \vee \neg \mathtt{x}_1) \wedge (\mathtt{y}_2 \vee \mathtt{x}_2) \wedge (\neg \mathtt{y}_2 \vee \neg \mathtt{x}_2).$$

In general, evaluating DQBFs is harder than evaluating QBFs. Peterson and Reif (1979) have shown, that evaluating DQBFs is NEXPTIME-complete. Furthermore, DQBFs can be used to solve the *partial equivalence checking* problem. The task of this problem is to extend a only partially defined logical circuit to match the functionality of a fully specified circuit (see Gitina, Reimer, et al. 2013).

To define the semantic of DQBF, one can use Skolem functions. Since this directly leads to the embedding of DQBF into HOL, we will forgo the definition of the semantic of DQBF and instead directly give the embedding into HOL. First we need the concept of a Skolem function.

Definition 3.3.2 (Skolem Function for DQBFs)**.** Let $\mathtt{t}$ be an arbitrary DQBF and $\mathtt{y}_i$ a existentially quantified variable of $\mathtt{t}$. Furthermore, let $D_{\mathtt{y}_i} = \{\mathtt{x}_{l_1}, \mathtt{x}_{l_2}, \ldots,$

$\mathtt{x}_{l_n}\}$ be the dependency set of $\mathtt{y}_i$ in $\mathtt{t}$ where $l_1 < l_2 < \cdots < l_n$. I.e. we fix a order of the universally quantified variables appearing in the dependency set. Then $\sigma_\mathtt{t}$ is the following mapping from existentially quantified variables to HOL terms:

$$\sigma_\mathtt{t} : \mathtt{y}_i \mapsto {}^\mathtt{t}sk^i_{\underbrace{o\to\cdots\to o}_{n \text{ times}}\to o}\, x_o^{l_1}\, x_o^{l_2}\, \ldots\, x_o^{l_n}.$$

The fresh function symbol ${}^\mathtt{t}sk^i$ is the *Skolem function* of $\mathtt{y}_i$ in $\mathtt{t}$.

Definition 3.3.3 (Translation from DQBF to HOL)**.** Let $\mathtt{t}$ be a DQBF in prenex form, then

$$\begin{aligned}&\rho(\forall \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n.\, \exists \mathtt{y}_1(\mathtt{x}_{l_1^1}, \ldots, \mathtt{x}_{l_1^{n_1}}), \ldots, \mathtt{y}_m(\mathtt{x}_{l_m^1}, \ldots, \mathtt{x}_{l_m^{n_m}}).\, \Phi) = \\ &\quad \exists\, {}^\mathtt{t}sk^1, \ldots, {}^\mathtt{t}sk^m.\, \forall x_o^1, x_o^2, \ldots, x_o^n.\, \rho'(\Phi)\end{aligned}$$

is the translation function into a HOL formula. The function ρ' translates the matrix and is defined recursively as:

$$\begin{aligned}\rho'(\mathtt{x}_i) &= x_o^i & \rho'(\mathtt{s} \vee \mathtt{t}) &= \rho'(\mathtt{s}) \vee_{o\to o\to o} \rho'(\mathtt{t}) \\ \rho'(\mathtt{y}_i) &= \sigma_\mathtt{t}(\mathtt{y}_i) & \rho'(\mathtt{s} \wedge \mathtt{t}) &= \rho'(\mathtt{s}) \wedge_{o\to o\to o} \rho'(\mathtt{t}) \\ & & \rho'(\neg \mathtt{s}) &= \neg_{o\to o} \rho'(\mathtt{s})\end{aligned}$$

In other words, we translate a DQBF into HOL by replacing existentially quantified variables with functions that calculate satisfying assignment to the existentially quantified variables from the universally quantified variables. A DQBF is satisfiable if such functions exist. Therefore, a DQBF $\mathtt{t}$ is true if and only if $\rho(\mathtt{t})$ is valid.

Systems for DQBF Solving

Recently some systems which solve DQBF formulas have emerged. For a relatively recent overview see Kovásznai (2015).

A first attempt at developing an efficient general purpose DQBF solver has been made by Fröhlich, Kovásznai, and Biere (2012). Their approach was to develop DPLL-style search based algorithm and to lift various successful techniques from SAT solving to DQBF solving. The authors, however, concluded that «it does not perform very well».

Another approach emerges from the observation that Effectively Propositional Logic (EPR), a fragment of first-order logic, is also NEXPTIME-complete. A solving approach for EPR formulas is the *Inst-Gen* calculus. This calculus combines unification to generate clause instantiations with an abstraction refinement loop. The *i*DQ system[7] (Fröhlich et al. 2014) implements this calculus

7 available at: `http://fmv.jku.at/idq/`

with specific low level optimizations for the Boolean domain, such as bit mask operations for unification. This system was the first publicly available solver for DQBF.

The HQS system (Gitina, Wimmer, et al. 2015) is a DQBF solver based on quantifier elimination. A universally quantified variable x can be removed from the dependency set of an existentially quantified variable by forming the conjunction of the formula where x is instantiated once with a constant which is always true and once with a constant which is false. The HQS system attempts to reduce the DQBF input problem to a QBF problem by systematically eliminating variables. To minimize the number of necessary expansions the HQS system utilizes a partial MAXSAT solver.

Furthermore, some preprocessing techniques have been lifted from QBF solving to DQBF by Wimmer, Gitina, et al. (2015) and subsequently been implemented in DQBF preprocessing tool for HQS (Wimmer, Reimer, et al. 2017).

4 Preprocessing Techniques

In this chapter we will introduce multiple preprocessing techniques for HOL which are inspired by QBF and DQBF preprocessing techniques.

As outlined in section 3.2, modern QBF solving systems employ various preprocessing tools. Thus, we give an overview of preprocessing techniques used in QBF and DQBF solving in section 4.1, before moving to HOL in the subsequent section.

Section 4.2 discusses universal reduction in the context of HOL. We already introduced this method as part of Q-resolution in the introduction chapter and will describe its extension to HOL.

Subsequently, section 4.3 will discuss ways to detect constant literals. In the context of QBF solving a literal is constant, if it can be replaced with a constant in all clauses. Pure literals, i.e. literals which only appear with one polarity are constant. In the context of HOL we will add constants as unit clauses.

One of the central preprocessing techniques for QBF is Quantified Blocked Clause Elimination (QBCE). The preprocessing system BLOQQER is named after this preprocessing method (cf. Biere, Lonsing, and Seidl 2011). While we have a look at QBCE in the context of QBF solving as part of section 4.1, we will also discuss an adaption to HOL in section 4.4.

Finally, in section 4.5 we will present a preprocessing technique which is not an adaption of a QBF, or DQBF preprocessing technique. Instead, we take inspiration from the encoding of QBF formulas into first-order logic and hypothesize that we can adapt this encoding to accelerate reasoning with Boolean literals by delaying primitive substitution.

The discussion in the chapter is a theoretical one, but we also implemented all the techniques presented here as part of the Leo-III theorem prover. This implementation alongside an empirical evaluation is discussed in the next chapter.

4.1 Preprocessing for QBF and DQBF

In this section we give an overview of current preprocessing techniques used for QBF and DQBF problems. Instead of attempting to give a full survey over all

available preprocessing techniques, we rely on two useful sources.

The first source is the description of QRAT by Heule, Seidl, and Biere (2014). The QRAT system is used to represent poofs of deductions conducted during preprocessing of QBF problems. After concluding the preprocessing process, the generated QRAT proofs can be verified by a special proof checking tool. At the heart of QRAT are three Quantified Resolution Asymmetric Tautology rules. According to the authors, those three rules can be used to «efficiently express *all* preprocessing techniques used in state-of-the-art preprocessors». Therefore, this publication necessarily gives an overview of most currently used preprocessing techniques for QBF solving.

Secondly, we rely on the lifting of multiple QBF preprocessing techniques to DQBF preprocessing by Wimmer, Gitina, et al. (2015). Not only are these techniques good candidates for further lifting to HOL, their presentation and categorization is also very insightful.

Preprocessing Techniques in the Context of QRAT

In the presentation of QRAT Heule, Seidl, and Biere (2014) distinguished between three types of preprocessing rules. *Clause Elimination Rules* are used to remove clauses, while preserving unsatisfiability. *Clause Modification Rules*, on the other hand, changes clauses by adding or removing literals. Finally, *Clause Addition Rules* are used to add new clauses to the problem.

A rule in the first category is the elimination of clauses which are tautological. Clauses which contain a literal and its negation at the same time are tautological and can be removed. Furthermore, a clause which is a subset of another clause is subsumed by that clause and can be removed. Clauses consisting of only an $\exists$-literal are also superfluous and can be removed safely. Finally, clauses which fulfill certain properties are called *blocked*. Removal of these clauses is the *(Quantified) Blocked Clause Elimination* (QBCE) technique. Since this technique is very successful, we will discuss it separately later.

As discussed above, universal reduction allows the removal of certain $\forall$-literals from clauses. Hence, this is a clause modification rule. Moreover, if the problem contains two clauses $C \vee l$ and $D \vee \bar{l}$ where $C \subseteq D$, then after applying Q-resolution the resolvent subsumes C. Using this insight, the *strengthening* rule modifies C by removing l. The *Unit Literal Elimination* rule can be used to remove clauses containing l and occurrences of $\bar{l}$ if l is an $\exists$-literal and occurs as a clause by itself and *Universal Pure Literal Elimination* removes $\forall$-literals if it occurs in only one polarity. *Covered Literal Addition* adds literals that occur in all non-tautological resolvents of a clause to that clause. *Equivalence Replacement* replaces literals with literals which are equivalent with them.

Two literals l and k are equivalent, if the two clauses $l \vee \bar{k}$ and $\bar{l} \vee k$ are part of the problem.

The two clause addition rules both remove variables. *Variable Elimination* removes an existentially quantified variable, by replacing clauses containing this variables by their non-tautological resolvents. Secondly, the *Universal Expansion* rule removes innermost $\forall$-literals by duplicating and modifying clauses.

Preprocessing Techniques Lifted to DQBF

The difference between QBFs and DQBFs is that DQBFs admits quantifier relations which do not form a total order. This translates to preprocessing techniques. Those QBF preprocessing techniques, where the implicit dependency («left of») can be substituted by an explicit dependency are good candidates for a translation to DQBF.

When describing preprocessing for DQBF, Wimmer, Gitina, et al. (2015) start with *backbones*, *monotonic*, and *equivalent* variables. Backbone variables are variables that will result in an unsatisfiable DQBF when replaced with a constant. And for a DQBF ϕ a variable $\mathtt{v}$ is monotonic, if either $\phi[\mathbb{F}/\mathtt{v}] \wedge \neg\phi[\mathbb{T}/\mathtt{v}]$ or $\phi[\mathbb{T}/\mathtt{v}] \wedge \neg\phi[\mathbb{F}/\mathtt{v}]$ is unsatisfiable. Both definitions are generalizations of unit and pure literals as discussed before and either indicate an unsatisfiable problem, or can be replaced by a constant. In fact, unit literals or pure literals are two purely syntactic criterions for detecting backbone or monotonic variables, respectively. A further criterion utilizes the binary implication graph. This is the graph formed by taking the literal as nodes and adding edges for all implications $l \to m$ in the problem[1]. Not only can this graph be used to give some syntactic criterion for backbone and monotonic literals, it can also be used to find equivalent literals. Furthermore, the authors mention, that SAT solvers can be utilized to find backbone and monotonic literals. While this might be untypical for QBF and DQBF solving, we investigated this approach since HOL problems typically have much fewer and smaller clauses.

The second type of DQBF preprocessing techniques reduces the dependency sets of variables. Again a sufficient criteria is given. This criterion utilizes the *variable-clause incidence graph*, which is a bipartite graph with edges from variables to the clauses they appear in. An existential variable x then is independent of a universal variable y if there is no path from x to y visiting only existentially quantified variables which are dependent on the universal variable y.

Techniques resulting from proof search procedures for QBF form the third set of DQBF preprocessing techniques. This includes universal reduction, res-

1 This especially means that a clause of the form $\mathtt{x} \vee \mathtt{y}$ induces four edges: $(\mathtt{x}, \neg\mathtt{y})$, $(\neg\mathtt{x}, \mathtt{y})$, and the reverse edges.

olution and universal expansion. While universal reduction and resolution together form a complete calculus for QBF, this is no longer true for DQBF. Nevertheless, resolution can be used under some condition to eliminate existential variables. Universal expansion is used to eliminate a universal variable by duplicating the clauses and replacing the variable with the constants $\mathbb{T}$ and $\mathbb{F}$.

Then the authors present blocked clause elimination for DQBF. They also generalize then notion of hidden and covered literals, which are literals that can be added to clauses without changing the status of the problem. Adding them will increase the chance that a clause is blocked and can removed.

Finally, the structure extraction is discussed. In practice, DQBF problems are often generated from Boolean circuits or more complex expressions by clausification. Some solvers do not rely on a problem in CNF. In this case, structure extraction can be used to recover the original structure of the problem.

Blocked Clause Elimination

Blocked Clause Elimination (BCE) is one of several preprocessing techniques which utilize *blocked clauses*. The concept of blocked clauses has its origins in SAT solving (cf. Järvisalo, Biere, and Heule 2010). It was then lifted to QBF solving by Biere, Lonsing, and Seidl (2011) and to DQBF by Wimmer, Gitina, et al. (2015).

Beside eliminating blocked clauses, it is also possible to carefully add small blocked clauses. This technique is called *blocked clause addition* and has been shown to be useful in certain situations (Järvisalo, Heule, and Biere 2012).

We now give definitions of blocked clause as used by Wimmer, Gitina, et al. (2015) for DQBF. A clause is blocked if it contains an $\exists$-literal such that all resolvents on that literal are tautological.

Definition 4.1.1 (Blocked Clause)**.** Let Φ be a DQBF, C a clause in Φ, and $l \in C$ a $\exists$-literal. The literal l is a *blocking literal* for C if for all $C' \in \Phi$ with $\bar{l} \in C'$ there is a literal m such that $\{m, \bar{m}\} \in (C \cup C') \setminus \{l, \bar{l}\}$ and $D_m \subseteq D_l$. A clause is *blocked* if it contains a blocking literal.

Blocked clauses can be removed from problems without changing the satisfiability status of the problem. As this is a purely syntactic criterion, it is straightforward to search for blocked clauses. To increase the chance of a clause being blocked, hidden literals can be added. A literal $l \notin C$ is a *hidden literal* for C if there is a clause $\{l_1, \ldots, l_n, \bar{l}\} \in \Phi$ such that $\{l_1, \ldots, l_n\} \subseteq C$. Then replacing C by $C \cup \{l\}$ results in a satisfiability equivalent problem. Another type of literal which can be added to clauses without changing the status of the problem

are *covered literals*. The definition of covered literals in the case DQBF is a quite technical and we instead quote a rough intuition given by Wimmer, Gitina, et al. (2015):

> If a literal k is already contained in all non-tautological resolvents of a clause C with pivot literal l, then k may be added to C resulting in an equivalent formula.

Recently Kiesl et al. (2017) adapted the concept of blocked clause to first-order logic. The authors faced the challenge of handling unification and equality. When analyzing blocked clause elimination in the context of higher-order logic, the same difficulties arise. We discuss this generalization together with additional challenges arising in HOL in section 4.4.

4.2 Universal Reduction

In section 3.1 we already introduced universal reduction as part of the Q-resolution rule. Universal reduction is necessary for the completeness of Q-resolution.

Assuming a problem in CNF, we can describe universal reduction as defined by Wimmer, Gitina, et al. (2015) for DQBF problems as an explicit rule:

$$\frac{l \vee j_1 \vee \cdots \vee j_n \qquad l \text{ is a } \forall\text{-literal, } \mathrm{var}(l) \notin D_{\mathrm{var}(j_i)}}{j_1 \vee \cdots \vee j_n}$$

The meaning of the rule is: Given a clause and a $\forall$-literal, if none of the existentially quantified variables in that clause depend on the variable of that literal, then the literal can be removed from the clause. This is quite intuitive. If all the other variables appearing in the clause are independent of the $\forall$-literal, then this literal can just be set to false and therefore become irrelevant for the truth value of the clause.

In skolemized HOL formulas the dependencies are explicitly present in the terms. All the universally quantified variable an existentially quantified variable depends on appear as arguments to the Skolem functions. Furthermore, a more complex term depends on a universally quantified variable if the variable appears free in that term.

Hence, the universal reduction rule for HOL is[2]:

2 This rule is slightly more restrictive that it needs to be. If the literal in question appears multiple times, the rule can not be applied anymore. We can, however, assume that duplicated literals have been removed during clausification.

$$\frac{[x_o]^\alpha \vee [s_1]^{\beta_1} \vee \cdots \vee [s_n]^{\beta_n} \qquad x_o \notin \mathcal{F}_{s_i}}{[s_1]^{\beta_1} \vee \cdots \vee [s_n]^{\beta_n}} \mathcal{R}(UR)$$

Since $n \geq 1$, this rule can not derive the empty clause, even though the clause containing only a singular Boolean variable is certainly counter-satisfiable. Applying this rule does not change the validity status of the problem:

Theorem 4.2.1 (Soundness of Universal Reduction)**.** Let S be a skolemized HOL formula. Then S is valid if and only if S' is. Where S' is S after applying $\mathcal{R}(UR)$ to a clause $C \in S$.

Proof. Let C' be the result of applying the rule to C and $l = [x_o]^\alpha \in C$ the removed literal. We have that $C' \subset C$.

Assume S is counter-satisfiable. Then there is an interpretation $\mathcal{M}$ and a variable assignment σ such that $\|S\|^{\mathcal{M},\sigma} = \mathbb{F}$. Since S is of the form $D_1 \wedge \cdots \wedge D_n \wedge C \wedge D_{n+1} \wedge \cdots \wedge D_m$, we either have that $\|C\|^{\mathcal{M},\sigma} = \mathbb{F}$, or that $\|C\|^{\mathcal{M},\sigma} = \mathbb{T}$. In the first case, $\|C'\|^{\mathcal{M},\sigma} = \mathbb{F}$ since all terms in C are false and therefore so are the terms in C'. In the second case, some other clause D in S and S' is false and the status of C' does not matter. We can conclude, that interpretation and variable assignment also falsifies S'.

Assume S' is counter-satisfiable and again let $\mathcal{M}$ and σ be the falsifying interpretation and variable assignment. The only non-trivial case is if C' is the only clause falsified by this assignment and $\|l\|^{\mathcal{M},\sigma} = \mathbb{T}$. We can define:

$$\sigma' : v \mapsto \begin{cases} \mathbb{F}, & \text{if } v = x_o \text{ and } \sigma(x_o) = \mathbb{T} \\ \mathbb{T}, & \text{if } v = x_o \text{ and } \sigma(x_o) = \mathbb{F} \\ \sigma(v), & \text{otherwise.} \end{cases}$$

Then, by definition $\|l\|^{\mathcal{M},\sigma'} = \mathbb{F}$ and, since x_o does not appear free in any other term of C, we also get $\|C\|^{\mathcal{M},\sigma'} = \mathbb{F}$. This flip might also change to truth value of other clauses in C too, but this does not change the conclusion that s is also counter-satisfiable. □

Listing 2 is a pseudo code implementation of the universal reduction rule. The polarity ($\mathrm{pol}(l)$) and variable ($\mathrm{var}(l)$) of HOL literals is defined analogously to the QBF case.

4.3 Constant Extraction

Two simple syntactic rules which are applied in QBF and DQBF preprocessing are the detection and elimination of pure and unit literals.

```
1  function UniversalReduction(M)
       Input: A set M of skolemized clauses.
       Output: A set R of clauses, where some contain fewer literals.
2      R ← ∅
3      foreach C ∈ M do
4          C′ ← C
5          B ← {l | l ∈ C, l is either x^i_o or ¬x^i_o.}
6          P ← {(var(l), pol(l)) | l ∈ B.}
7          foreach l ∈ B do
8              if (var(l), ¬ pol(l)) ∈ P then
                   // Clause is tautological and can be removed.
9                  Continue in line 3.
10             else
11                 F ← ∪_{l′∈C, l′≠l} F_l
12                 if var(l) ∉ F then
13                     C′ ← C′ − l
14         R ← R ∪ {C′}
15     return R
```

Listing 2: Universal Reduction for HOL.

A pure literal is a literal which only occurs in one polarity in the whole problem. Since existential pure literals can be replaced by the true constant, all clauses containing pure ∃-literals can be removed. Pure ∀-literals, on the other hand, can be removed from the clauses they appear in.

A literal l is a unit literal, if there is a clause which contains l and no other literals. Hence, unit literals are literals which occur alone in a clause. Accordingly, clauses which contain only one literal are called *unit clauses*. The variable in a unit ∃-clause can be replaced by a constant and problems containing unit ∀-literals are unsatisfiable.

As described in section 4.1 backbone and monotonic variables are a more general and semantic variant of unit and pure literals.

Pure Literal Detection via Unification Tests is Unsound

A first attempt of implementing pure literal detection for HOL might be along the following lines:

Given an input formula s_o in clause form and a literal l appearing in this

problem. Iterate over all clauses C and literals l' in C. Continue if $l = l'$. Otherwise, check if $\mathrm{pol}(l) = \neg\,\mathrm{pol}(l')$. If this is the case l is not pure, nor is l'. Finally, check if l and l' are unifiable. Since unification for HOL is undecidable, this step must be an over approximation. If the two literals are possibly unifiable, neither l nor l' are pure. If no counterexample for pureness is found, l is pure and can be added as a unit clause.

This procedure, however, is unsound in the presence of equality. Consider for example the problem $(f_{o\to o} = \lambda x.\,\neg x) \wedge (f_{o\to o}\,(g_{\iota\to o}\,a)) \wedge (g_{\iota\to o}\,a \vee x_o)$. In this case, $g_{\iota\to o}\,a$ is not unifiable with any literal in the first two clauses and therefore could be added as a unit clause. This results in $(f_{o\to o} = \lambda x.\,\neg x) \wedge (f_{o\to o}\,(g_{\iota\to o}\,a)) \wedge (g_{\iota\to o}\,a \vee x_o) \wedge (g_{\iota\to o}a)$. Note that the original problem is satisfiable by setting $x_o = \mathbb{T}$ and $g_{\iota\to o}\,a = \mathbb{F}$. The new one, however, is no longer satisfiable, since $(\lambda x.\,\neg x)\,(g_{\iota\to o}a)$ β-reduces to $\neg(g_{\iota\to o}\,a)$.

Overall, this approach fails because HOL formulas contain equality symbols which carry semantical meaning without being explicitly axiomatized in the input problem. Furthermore, the arbitrary nesting of function symbols and operators is an additional challenge which must be handled by a constant extraction algorithm.

4.3.1 SAT Based Constant Extraction

As discussed in section 4.1 Wimmer, Gitina, et al. (2015) mention SAT solving as a procedure for detecting backbone and monotonic variables. Such a procedure was described by Pigorsch and Scholl (2010). We will first describe their algorithm, before describing our adaption to HOL.

Given a QBF formula t in CNF. That is, t has the form $Q_1\mathtt{x}_1 \ldots Q_n\mathtt{x}_n.\,s$ where s is a conjunction of clauses. If for any literal l it holds that $s \to l$, then $Q_1\mathtt{x}_1 \ldots Q_n\mathtt{x}_n.\,s \wedge l$ is equivalent to t. In this case l is a *constant literal*. Since variables which are existentially quantified in the QBF problem t are implicitely universally quantified in $s \to l$ this procedure is only approximative. The validity of $s \to l$ is equivalent to the unsatisfiability of $s \wedge \neg l$. Hence, a simple procedure to search for constant literals in a problem with n variables could make $2 \times n$ calls to a SAT solver to test for unsatisfiability. One call for each propositional variable and one call for the negated variable.

This can be optimized. Should a problem handed to the SAT solver be satisfiable – which will be the case for all literals which are not constant – the solver returns a model for the formula. This model can be used to avoid future checks. Given a model for $s \wedge l_i$ which assigns $\mathbb{T}$ to the literal l_j $(i \neq j)$, then this model will also be a model for $s \wedge l_j$. Hence, the algorithm can be improved by maintaining a queue of literals yet to be checked. If a generated

problem is satisfiable with a model which makes a literal l true the literal $\bar{l}$ can be removed from the queue.

Furthermore, some capabilities of modern SAT solvers can be exploited. Before doing the unsatisfiability checks, a model for s by itself can be computed. After removing literals from the queue based on this model as described before, the SAT solver can be introduced to try the yet unused value first when guessing assignments for variables. This increases the chance to remove literals from the queue after the first check. Lastly, many recent SAT solvers support incremental solving, which allows for the addition and removal of temporary assumptions without the necessity to restart the whole reasoning process. All those optimizations have been implemented in our adaption of the algorithm.

Adapting SAT Based Constant Extraction to HOL

Our adaption to HOL is quite straightforward. In a first step we map all HOL literals to SAT literals, ensuring that syntactically equal HOL terms get the same literal assigned. Then we invoke the SAT solver as described above.

This will result in a set of constant SAT literals, which then can be mapped back to HOL literals and added to the original input formula. To capture some relations between equality terms we add some transitivity constraints to the generated SAT problem. Before we describe this procedure, we give a more formal definition of the translation to SAT:

Definition 4.3.1 (SAT Literal Mapping)**.** A SAT literal mapping is a function $\lfloor . \rfloor$ from a set of HOL terms to proportional literals, such that:

1. If $\lfloor S \rfloor$ is defined for a HOL term S, then so is $\lfloor \neg S \rfloor$ and $\overline{\lfloor S \rfloor} = \lfloor \neg S \rfloor$.
2. If $\lfloor S = T \rfloor$ is defined for HOL terms S and T, then so is $\lfloor S \neq T \rfloor$ and $\lfloor S = T \rfloor = \overline{\lfloor S \neq T \rfloor}$.
3. If $\lfloor S = T \rfloor$ is defined, then so is $\lfloor T = S \rfloor$ and $\lfloor T = S \rfloor = \lfloor S = T \rfloor$.
4. If $\lfloor S \neq T \rfloor$ is defined, then so is $\lfloor T \neq S \rfloor$ and $\lfloor T \neq S \rfloor = \lfloor S \neq T \rfloor$

Overall, we require that negation behaves as expected, and that the order of terms in equations does not matter. The first requirement can be implemented by maintaining a directory which maps HOL terms to SAT variables and by appropriately removing and adding negations. The second requirement can be fulfilled by ordering the terms consistently before translation. This can be done by any total order. We choose a simple lexicographic order on terms.

We can now describe the translation from HOL into SAT:

Definition 4.3.2 (SAT Mapping of Formulas)**.** Given a SAT literal mapping $\lfloor . \rfloor$ and a HOL formula S in clause form. I.e. S is of the form $C_1 \wedge C_2 \wedge \cdots \wedge C_n$, where C_i are clauses and the explicit quantifiers have been removed by Skolemization. Then the SAT mapping $\langle S \rangle$ is $\{\lfloor l \rfloor \mid l \in C_1\} \wedge \cdots \wedge \{\lfloor l \rfloor \mid l \in C_n\}$.

Note that the mapping removes all the quantifiers from the input formula. This SAT mapping has the desired property:

Theorem 4.3.1 (Soundness of SAT Mapping)**.** Given a HOL formula S of the form as described above and a SAT literal mapping $\lfloor . \rfloor$. If $\langle S \rangle \rightarrow \lfloor l \rfloor$ is valid for an arbitrary literal l, then S and $S \wedge l$ are satisfiability equivalent.

To prove this, we first prove this lemma:

Lemma 4.3.2. Given a SAT literal mapping $\lfloor . \rfloor$ and a HOL formula S in clause form. Then if S is satisfiable if and only if $\langle S \rangle$ is satisfiable.

Proof. Assume that S is satisfiable and let $\mathcal{M}$ be an interpretation and σ a variable assignment such that $\|S\|^{\mathcal{M},\sigma} = \mathbb{T}$.

We use this interpretation and assignment to construct a model $\mathtt{I}$ for $\langle S \rangle$. Let $\mathtt{I}$ be the mapping from propositional variables to truth values, such that $\mathtt{I}(\mathtt{x}_i) = \|l\|^{\mathcal{M},\sigma}$ if $\lfloor l \rfloor$ is a variable $\mathtt{x}_i$ and $\|\mathtt{x}_i\|^{\mathtt{I}} = \neg\|l\|^{\mathcal{M},\sigma}$ if $\lfloor l \rfloor$ is of the form $\neg\mathtt{x}_i$. Since $\lfloor \neg l \rfloor = \neg\lfloor l \rfloor$ this is well-defined.

For each clause $C \in S$, there is a literal l such that $\|l\|^{\mathcal{M},\sigma} = \mathbb{T}$. By definition $\|\lfloor l \rfloor\|^{\mathtt{I}} = \mathbb{T}$ and hence for each proportional clause there is a mapped literal which the valuation $\|.\|^{\mathtt{I}}$ is true. Therefore, $\|\langle S \rangle\|^{\mathtt{I}} = \mathbb{T}$.

The second statement directly follows from the first one by negation. □

We can now proof theorem 4.3.1:

Proof. First note that $\langle S \rangle \rightarrow \lfloor l \rfloor$ is equivalent to $\neg(\langle S \rangle \wedge \neg\lfloor l \rfloor)$ and, by the definition of the mappings, to $\neg\langle S \wedge \neg l \rangle$. Therefore, $\langle S \rangle \rightarrow \lfloor l \rfloor$ is valid if and only if $\langle S \wedge \neg l \rangle$ is unsatisfiable and by lemma 4.3.2 the validity of $\langle S \rangle \rightarrow \lfloor l \rfloor$ implies unsatisfiability of $S \wedge \neg l$.

Assume S is satisfiable, but $S \wedge l$ is not. Hence, we know that for all interpretation $\mathcal{M}$ and a variable assignment σ such that $\|S\|^{\mathcal{M},\sigma} = \mathbb{T}$ we have that $\|l\|^{\mathcal{M},\sigma} = \mathbb{F}$. This implies that $\|\neg l\|^{\mathcal{M},\sigma} = \mathbb{T}$ and therefore $\|S \wedge \neg l\|^{\mathcal{M},\sigma} = \mathbb{T}$. Since at least one such interpretation exists this is a contradiction to the remark above.

The other direction is trivial, since all subsets of a satisfiable set of clauses are satisfiable. □

Adding Transitivity Constraints

The translation procedure just described disregards most of the structure of the formula. Only the basic propositional structure is retained. We now describe a simple procedure to add some additional structural information to the generated SAT problem.

Some literals in the generated SAT problem encode equalities between terms. The equality relation is transitive. Consider for example the formulas $s = t$ and $t = u$, then the transitivity of equality implies that $s = u$ holds. That is $s = t \wedge t = u \to s = u$ is a valid HOL formula. Since equality is implicitly handled by the Leo-III calculus, those clauses do not appear in the clause set during reasoning. They, however, can be added to the generated SAT problem as additional clauses.

A naïve procedure to do so would be to first collect all SAT literals which encode equalities an equality set $\mathcal{E}$ and then add all possible transitivity constraints to the SAT problems. The generated transitivity constraints would be formulas of form

$$\lfloor S_1 = S_2 \rfloor \wedge \lfloor S_2 = S_3 \rfloor \wedge \cdots \wedge \lfloor S_{k-1} = S_k \rfloor \to \lfloor S_1 = S_k \rfloor$$

where $\lfloor S_i = S_j \rfloor \in \mathcal{E}$. Those formulas can be translated to equivalent clauses which can be added to the problem:

$$\neg\lfloor S_1 = S_2 \rfloor \vee \neg\lfloor S_2 = S_3 \rfloor \vee \cdots \vee \neg\lfloor S_{k-1} = S_k \rfloor \vee \lfloor S_1 = S_k \rfloor.$$

This naïve approach, however, leaves much to be desired. Given the unrestricted length of the constraints they might have an arbitrary, even infinite, length. A more goal-oriented approach is needed for this problem. Efficient ways to create transitivity constraints have been proposed by Bryant and Velev (2000) which we adapted to the problem at hand and implemented.

In a first step, the set of equalities is represented as an undirected graph $G = (V, E)$. The set of nodes V is the set of all HOL terms[3], and $\{S, T\} \in E$ if $\lfloor S = T \rfloor \in \mathcal{E}$.

The authors then show that the set of all transitivity constraints is satisfied by those interpretations which satisfy the transitivity constraints induced by the chord-free cycles in the graph. A chord free cycle is a cycle $[v_1, v_2, \ldots, v_k, v_1]$ where $\{v_i, v_j\} \notin E$ for any $j \neq (i+1)$, except $i = k$ and $j = 1$. The transitivity constraint induced by such a cycle is $\lfloor v_1 = v_2 \rfloor \wedge \cdots \wedge \lfloor v_{k-1} = v_k \rfloor \to \lfloor v_1 =$

3 Since the procedures by Bryant and Velev (2000) consider literals which encode equalities abstractly, their set of nodes is a bounded set of natural numbers.

$v_k\rfloor$. Unfortunately, the number of chord-free cycles can be exponential in the number of edges in the graph.

By adding additional derived equality literals this can be avoided. To do so, the graph must be made *chordal*. A graph is chordal, if no cycle with more than three nodes is chord-free. Making a graph chordal by adding a minimal amount of additional edges is NP-complete, but good heuristic solutions exists. We implemented the heuristic presented by Bryant and Velev (2000).

The heuristic starts with $G_0 = G$ and iteratively constructs $G_i = (V_i, E_i)$ from G_{i-1} by removing a node from G_{i-1} and adding edges. If in step i the node v_i is removed, edges are added to G_i for each two nodes j, k where $\{j, v_i\} \in E_{i-1}$, $\{k, v_i\} \in E_{i-1}$, but $\{j, k\} \notin E_{i-1}$. This procedure is iterated until all nodes have been deleted. The final graph is G plus all edges created during the procedure.

The Entire Algorithm

We now give a pseudo code description of the complete algorithm. The presentation is split into two parts. Listing 3 shows the overall skeleton of the algorithm, while listing 4 contains the generation of the transitivity constraints from an equality graph.

The algorithm uses a mapping from tuples of terms to propositional variables. This mapping allows the representation of both equational and ordinary literals. A tuple (l, j) represents the equation $l = j$. To represent ordinary literals their partner is set to $\mathbb{T}$. Hence, the tuple $(l, \mathbb{T})$ is used to represent $l = \mathbb{T}$, which is equivalent to just l. In fact, the algorithm will generate literals of the form $[l_o = \mathbb{T}]^\alpha$ instead of $[l_o]^\alpha$. These two forms are equivalent and the first form is very similar to the internal representation used in Leo-III, while also simplifying the pseudo code.

We assume that the mapping from term tuples to propositional variables reorders the terms before requesting the variable. Hence, $M(a, b) = M(b, a)$. This removes the necessity to handle terms of the form $S = T$ and $T = S$ differently. Any total order can be used.

Assuming a literal in the SAT solver means adding this literal to the SAT problem as a until clause for the next call to solve. The SAT solver then automatically removes the assumed literal after finishing solving the problem[4]. Furthermore, asking the SAT solver for a model returns the set of SAT *literals* set to $\mathbb{T}$.

Line 29 in listing 3 uses a shorthand notation for expressing the polarity of a HOL literal. Here $[l]^{\mathbb{T}}$ means $[l]^{tt}$ and $[l]^{\mathbb{F}}$ means $[l]^{ff}$.

4 This is exactly the behavior of *pico*SAT, which was used in our implementation.

```
1  function ConstantExtraction(M)
       Input: A set M of skolemized clauses.
       Output: A set U of new unit clauses.
2      M ← empty map from term tuples to propositional variables.
3      S ← new incremental SAT solver instance.
4      E ← new equality graph.
5      foreach C ∈ M do
6          C′ ← ∅
7          foreach [l]^α ∈ C do
8              if l is of the form s = t then
9                  T ← (s, t)
10                 Add nodes s, t and the edge {s, t} to E.
11             else
12                 T ← (l, 𝕋)
13             V ← M(T)
14             if V not defined then
15                 V ← fresh SAT variable.
16                 M(T) ← V
17             Add V to C′ if α = tt and ¬V otherwise.
18         Add C′ to SAT solver S.
19     Add TransitivityConstraints(E, M) to S.
20     if Solve(S)=UNSAT then
21         return Input is contradictory.
22     L ← {l̄ | l ∈ Model(S)}
23     Introduce S to try l̄ first for l ∈ Model(S).
24     while L ≠ ∅ do
25         l ← an element from L.
26         Assume l in S.
27         if Solve(S)=UNSAT then
28             (s, t) ← M^{-1}(var(l))
29             N ← [s = t]^{¬ pol(l)}
30             if {N} ∉ M then
31                 U ← U ∪ {N}
32         else
33             L ← L \ Model(S)
34     return U
```

Listing 3: SAT Based Constant Extraction.

The last missing piece is the generation of the transitivity constraints given in listing 4. As described above the graph is first made chordal. Our presentation is simplified and will add edges already in the graph. Since a chordal graph has no chord-free cycles of length greater than three, enumerating the chord free cycles is trivial.

```
1 function TransitivityConstraints(E, M)
    Input: An equality graph E = (v, e), a mapping M from term tuples to
           propositional variables.
    Output: A set T of new propositional clauses.
2   v' ← v
3   while v' ≠ ∅ do
4     n ← node from v'
5     v' ← v' − n
6     e ← e ∪ {{s, t} | {n, s} ∈ e, {n, t} ∈ e}
7   return {M(a, b), M(a, c), ¬M(c, d) | {a, b} ∈ e, {a, c} ∈ e, {c, d} ∈ e}
```

Listing 4: Generate Transitivity Constraints.

4.4 Blocked Clause Elimination

We now discuss Blocked Clause Elimination (BCE) for HOL. Blocked Clause Elimination is an efficient SAT preprocessing technique which was also successfully applied to QBF (as QBCE) and DQBF solving. A description of BCE and QBCE can be found in section 4.1.

Recently BCE was lifted to first-order logic by Kiesl et al. (2017). When doing so, equality requires specialized handling. Therefore, FOL clauses can either be *blocked* or *equality-blocked*. Every equality-blocked clause is also a blocked clause, but some blocked clauses are not equality blocked. For input problems which do not contain equalities general first-order BCE is applied, while for problems containing equalities, only equality-blocked clauses can be eliminated.

We lifted ordinary blocked clause elimination to HOL and implemented the resulting algorithm. The theoretical adaption from FOL is straightforward. The main additional challenge is the undecidability of unification in the case of HOL. Adapting the implementation is a bit more tricky, since indexes which track unification partners must be adapted.

In the next subsection we will describe ordinary and equality blocked clause

elimination for FOL based on Kiesl et al. 2017. Subsection 4.4.2 then describe the specialties emerging in the HOL case.

4.4.1 First-order Blocked Clause Elimination

Remember that a DQBF clause is blocked if there is a blocking literal in the clause. A literal l is a blocking literal, if all resolvents on l contain complementary literals k and $\bar{k}$ with $D_k \subseteq D_l$.

Blocked Clauses Without Equality

As mentioned above, BCE for FOL differs if equalities are present. We first assume that the FOL formula under consideration does not contain any equalities.

In the case of FOL, resolution involves unification. This translates to the definition of blocked clauses.

Definition 4.4.1 (l-resolvent)**.** Given two clauses $C = l \vee C'$ and $D = n_1 \vee \cdots \vee n_i \vee D'$ with $i > 0$, such that $l, \bar{n_1}, \ldots, \bar{n_i}$ are unifiable by an MGU σ. Then $C'[\sigma] \vee D'[\sigma]$ is the *l-resolvent* of C and D.

Definition 4.4.2 (Blocked Clause)**.** A clause C is *blocked* by a literal $l \in C$ in a FOL formula F if all l-resolvents of C with clauses in $F \setminus \{C\}$ are valid.

A l-resolvent R without equality is valid if and only if there is a literal k such that $\{k, \bar{k}\} \subseteq R$. Again blocked clauses can be removed from the problem without changing its satisfiability status.

Checking the validity of l-resolvents without further optimizations would require checking an exponential number of literal pairs per clause. This, however, can be reduced to a polynomial number of checks by using the algorithm described in listing 5.

Two further optimizations exist. First a index, such as a hash map, can be used to map literal to clause-literal pairs. A literal l is mapped to a pair (l', C) where l' has opposite polarity of l, $l' \in C$, and the predicate[5] of l and l' are the same. This index can be used to quickly retrieve all unification candidates. Furthermore, a queue of clause literal pairs is maintained. These represent clauses with potential blocking literals. The procedure first dequeues a clause-literal pair (C, l) from the queue and retrieves all resolution candidates D. Then the testing function is called. If the l-resolvent is found to be countersatisfiable,

5 In FOL predicates are the symbols which take terms to propositions. For example, in the proposition $P(a, f(b)) \wedge R(c)$ the symbols P and R are predicates. They correspond to functions of type $\tau \to o$ in the HOL world.

```
1  function TestLResolvValid(C, l, D, N)
       Input: Clause C = l ∨ C′ and D = N ∨ D′, where l̄ and n are unifiable for
              all n ∈ N.
       Output: 𝕋 if all l-resolvents of C and D are valid, 𝔽 otherwise.
2      foreach n ∈ N do
3          T ← {n}
4          while l is unifiable with all literals in T̄ := {t̄ | t ∈ T} do
5              σ ← MGU of l and all literals in T̄
6              K ← all pairs of complementary literals in C′[σ] ∨ (D \ T)[σ].
7              if K = ∅ then
8                  return 𝔽
9              if All pairs in K contain a literal n[σ] with n ∈ N then
10                 T ← T ∪ {n | n[σ] is part of a pair in K}
11             else
12                 break the while loop
```

Listing 5: Testing the Validity of l-resolvents.

the clause-literal pair (C, l) is deactivated and saved as being deactivated by D. Should, on a later point, D be found to be blocked and therefore be removed, then (C, l) is re-enqueued.

We adapted this algorithm to HOL. First, however, we discuss BCE with equalities for FOL.

Equality Blocked Clauses

When equality is added as a native symbol, blocked clause elimination becomes unsound. Consider the following example taken from Kiesl et al. (2017). Let $C := P(a)$ and $F := (a = b \wedge \neg P(b))$. In this example, $P(a)$ and $P(b)$ are not unifiable. Hence, C is blocked. However, F alone is satisfiable while $(P(a) \wedge a = b \wedge P(b))$ is not.

To solve this problem more clauses must be taken into account during the test. The algorithm must not only test clauses which are unifiable with the clause under consideration, but it must consider all those clauses which have a literal with the same head symbol and opposite polarity.

This is implemented by *flattening* a literal:

Definition 4.4.3 (Flattening)**.** Given a clause $C = L(t_1, \ldots, t_n) \vee C'$. Flattening the literal $L(t_1, \ldots, t_n)$ in C yields the clause: $\widetilde{C} = \bigvee_{1 \leq i \leq n} x_i \neq t_i \vee$

$L(x_1, \ldots, x_n) \vee C'$

The literal $L(x_1, \ldots, x_n)$ in $\widetilde{C}$ is the *flattened literal*.

Such a flattened clause is equivalent to $\bigvee_{1 \leq i \leq n} x_i = t_i \rightarrow L(x_1, \ldots, x_n) \vee C'$. Hence, flattening clauses in a problem does not change the satisfiability status of the problem. We can define the concept of flat resolvents based on flattening of clauses. Flat resolvents are resolvents generated by resolving between flattened clauses.

Definition 4.4.4 (Flat l-resolvent)**.** Given two clauses $C = l \vee C'$ and $D = n_1 \vee \cdots \vee n_i \vee D'$ with $i \geq 1$, and such that $l, \bar{n_1}, \ldots, \bar{n_i}$ have the same predicate symbol and polarity. Let $\widetilde{C}$ and $\widetilde{D}$ be the result of flattening the literals $l, n_1, \ldots, n_i$ in C and D and $\tilde{l}, \tilde{n}_1, \ldots, \tilde{n}_i$ are the flattened literals.

The resolvent

$$(\widetilde{C} \setminus \{\tilde{l}\})[\sigma] \vee (\widetilde{D} \setminus \{\tilde{n}_1, \ldots, \tilde{n}_i\})[\sigma]$$

of $\widetilde{C}$ and $\widetilde{D}$, with σ being an MGU of $\tilde{l}, \bar{\tilde{n}}_1, \ldots, \bar{\tilde{n}}_i$, is a *flat l-resolvent* of C and D.

Finding a MGU for flattened clauses becomes trivial. For the flat literals $L(x_1, \ldots, x_n), \bar{N}_2(y_{11}, \ldots, y_{1n}), \ldots, \bar{N}_i(y_{i1}, \ldots, y_{in})$ the substitution $\sigma : y_{ij} \mapsto x_j$ is already a MGU.

As in the case of FOL without equality, a clause C is equality blocked if all flat l-resolvents for a literal $l \in C$ are valid:

Definition 4.4.5 (Equality Blocked)**.** A clause C is *equality blocked* by a literal $l \in C$ in a FOL formula F if the predicate of l is not $=$ and all flat l-resolvents of C with clauses in $F \setminus \{C\}$ are valid.

Equality blocked clauses are redundant and can be removed from the formula without changing its satisfiability status. Checking validity in the presence of equality is more complex. E.g. clauses containing literals such as $x = x$ are already valid.

To detect the validity of flat l-resolvents a congruence closure algorithm can be used. A flat l-resolvent R is valid if and only if the negation of its universal closure $\neg\forall R$ is unsatisfiable. Applying distributivity to the negation will result in an existentially quantified conjunction of negated literals. Skolemization will then replace the free variables of R with fresh constants. Hence, $\neg\forall R$ is equivalent to conjunction of ground equational literals.

The satisfiability of a conjunction of equalities between ground terms can be efficiently decided by congruence closure algorithms (see Shostak 1978). Kiesl

et al. (2017) note, that a complete congruence closure algorithm was to inefficient and they instead used a simplified version. First they ignored all equalities which were present in the clauses before flattening, secondly they only applied the equalities once, instead of recursively applying the congruence rule.

4.4.2 Higher-order Blocked Clause Elimination

We now discuss the adaption of ordinary blocked clause elimination to HOL. Therefore, we restrict ourselves to HOL problems without equality.

When lifting BCE to HOL, the main challenge is the undecidability of unification. This can be solved by restricting the used (blocking) literals to pattern literals. As described on page 20, unification between pattern literals is always decidable and if the literals are unifiable there is exactly one MGU. The definition of blocked clause, however, speaks about all resolvents. Hence, some additional care has to be taken to ensure the correct handling of literals which are not patterns. To do so we first introduce a few helpful terms.

Definition 4.4.6 (Head Symbol)**.** Given a HOL formula S_τ which has been β-normalized. S_τ is of the form $\lambda x^1_{\tau_1}. \ldots \lambda x^i_{\tau_i}. t \ldots$ where $i \geq 0$ (there might be no λ-binders) and t is either a constant, a locally bound variable, or a free variable. The symbol t is the *head symbol* of S_τ.

Since we are interested in handling literals, we work with formulas (terms of type o) which have being β-reduced as far as possible.

The head symbol can either be a *flex* or *rigid* head. Those terms are originally used in the context of higher-order unification. See subsection 2.3.1 for a description of higher-order unification.

Definition 4.4.7 (Flex Head)**.** Assume S_τ is a HOL term, then S_τ has a *flex head* if the head symbol of S_τ is a free variable.

Definition 4.4.8 (Rigid Head)**.** Assume S_τ is a HOL term, then S_τ has a *rigid head* if it does not have a flex head.

Two terms with different rigid heads will never be unifiable. Hence, this can be used as a simple, but incomplete, check for non-unifiability.

We now need to adapt the concept of an l-resolvent and an blocked clauses based on this definition. We restrict blocking literals to pattern literals. A unification test between a pattern and a non-pattern literal might still be non-terminating. After restricting the (potential) blocking literals l to pattern literals the following cases can occur when testing unification with another literal m:

1. The literal m is a pattern literal. Then unification is decidable and we proceed as in the FOL case.

2. The head of l is rigid and m is not a pattern literal:
   a) The head symbol of m is rigid and different from the head symbol of l. Then l and m are not unifiable.
   b) In all other cases l and m might be unifiable.

3. The head of l is flex and m is not a pattern literal. Then l and m might be unifiable.

The cases 2b and 3 are the critical ones. In those cases the we must assume the worst case. Furthermore, it is not enough to consider literals of opposite polarity to describe valid l-resolvents, since switching the polarity by adding an additional negation will still result in unifiable literals. Consider for example the two clauses $[f\,a]^{tt}$ and $[x\,a]^{tt}$ where f and a are constant symbols and x is a free function variable. Then this problem is equivalent to $[\neg(f\,a)]^{ff}$ and $[x\,a]^{tt}$. Now the unification problem $(\neg(f\,a)) = (x\,a)$ can be solved by the substitution which maps x to $\lambda y.\,\neg(f\,y)$ and the empty clause can be derived by resolution.

Such a transformation is only possible when one of the literals has a flex head. In this situation the negation symbol can be added to the rigid headed literal. When both literals have a rigid head, they are only unifiable if the head is the same and adding the negation literal would change this.

To capture this discussion we first define l-unification constraints.

Definition 4.4.9 (l-unification constraints)**.** Given literals $[l]^{\alpha}, [n_1]^{\beta_1}, \ldots, [n_i]^{\beta_i}$ and a set of equalities U with i elements. Then U is a l-unification constraint if for all i it holds that $l = n_i \in U$ if $\beta_i \neq \alpha$ and either $(\neg l) = n_i \in U$ or $l = (\neg n_i) \in U$ if $\beta_i = \alpha$.

Now we can define l-resolvents based on this definition.

Definition 4.4.10 (l-resolvent)**.** Let $C = [l]^{\alpha} \vee C'$ and $D = [n_1]^{\beta_1} \vee \cdots \vee [n_i]^{\beta_i} \vee D'$ be two clauses and U be a l-unification constraint on $l, n_1, \ldots, n_i$. Furthermore, assume that U is solvable by a MGU σ. Then $C'[\sigma] \vee D'[\sigma]$ is a *l-resolvent* of C and D.

The definitions so far did not capture the decidability of the unification problem. We do this in our definition of blocked clauses.

Definition 4.4.11 (Pattern Blocked Clause)**.** A clause C is *pattern blocked* by a literal $l \in C$ in a HOL formula F if:

- l is a pattern literal.
- None of the clauses in $F - C$ contains a literal which is not a pattern and has a flex head.
- All non-pattern literals in the clauses of $F - C$ have a different head symbol than l.
- All l-resolvents with clauses of $F - C$ are valid.

The first three conditions in definition 4.4.11 ensure that deciding if a clause is pattern blocked by a literal is always decidable. Those three conditions can easily be checked and if they are fulfilled all non-pattern literals can be excluded from unification checks, since they will never be unifiable with l.

We conjecture that pattern blocked clauses can be removed from HOL problems without changing its satisfiability status.

An Algorithm for Pattern Blocked Clause Elimination

We now describe an algorithm to find pattern blocked clauses in HOL problems. This algorithm corresponds to our implementation in Leo-III and is an adaption of the algorithm described by Kiesl et al. (2017)

The algorithm maintains four data structures. A pattern index P is used to retrieve potential pattern unification candidates. Secondly, a dictionary N maps head symbols together with their polarity to rigid non-patterns with the same polarity. While testing clauses and potential blocking literals, two further data structures are used. A priority queue Q collects clause literal pairs (C, l), where $l \in C$ and l is a pattern literal. Finally, a deactivation index D is used to track clauses which prohibit clause literal pairs from being blocked.

Since the pattern index must handle flex headed literals differently from rigid headed pattern literals it internally uses one map and two sets. The dictionary $P.rigidIndex$ is similar to the non-pattern dictionary N and maps head symbols and polarity to rigid patterns with the same polarity. The two sets $P.rigid$ and $P.flex$ collect all clause pattern literal pairs where the head of the literal is rigid and flex, respectively.

Listing 6 describes how fresh clause literal pairs are inserted into the pattern index. This is done by a simple case distinction on the type of head literal in the added clause literal pair has.

```
1 function AddToIndex(P, C, l)
      Input: A pattern index P, a clause C, and a pattern literal l ∈ C
2     h ← head of l.
3     p ← polarity of l.
4     if l is a flex literal then
5         Q.flex ← Q.flex ∪ {(C, l)}
6     else
7         Q.rigid ← Q.rigid ∪ {(C, l)}
8         Q.rigidIndex((p, h)) ← Q.rigidIndex((p, h)) ∪ {(C, l)}
```

Listing 6: Adding a clause literal pair to the pattern index.

The method described in listing 7 is used to query the pattern index for the possible resolution candidates. The candidates are grouped by clause to enable a fast retrieval of all potential candidates in any clause. If the query literal has a flex head, all saved literals are returned.

```
1 function QueryPatternIndex(P, l)
      Input: A pattern index P and a pattern literal l ∈ C
      Output: A set of tuple (C, L) where C is a clause and L ⊆ C, such that
              the elements of L are potentially unifiable with l.
2     h ← head of l.
3     p ← polarity of l.
4     if l is a flex literal then
5         S ← P.flex ∪ P.rigid
6     else
7         S ← P.flex ∪ P.rigidIndex((¬p, h))
8     D ← {c | (c, _) ∈ S}
9     return {(C, L) | C ∈ D, L = {l | (C, l) ∈ S}}
```

Listing 7: Retrieving potential resolution candidates from the pattern index.

The deactivation index D must maintain two dictionaries. One to save which clauses are certain clause deactivates and another one to remember by which clauses a clause is deactivated.

The priority queue Q orders clause literal pairs by the number of possible resolution candidates returned by querying the pattern index.

We now present the outer loop of the pattern blocked clause elimination algorithm. The pseudo code is split into two parts. Listing 8 describes the first

iteration over the input problem to initialize the data structures, while listing 9 is the test loop.

```
1  function PBCE(M)
       Input: A equality free set M of skolemized clauses.
       Output: A potentially smaller set of clauses.
2      P ← empty pattern index. N ← empty dictionary of non-patterns.
3      φ ← ∅
4      foreach C ∈ M do
5          foreach l ∈ C do
6              if l is a pattern then
7                  AddToIndex(P,C,l)
8                  φ ← φ ∪ {(C, l)}
9              else
10                 p ← polarity of l. h ← head of l.
11                 if l is rigid. then
12                     N((p, h)) ← N((p, h)) ∪ {(C, l)}
13                 else
14                     if f is defined then
15                         return Input contains two flex non-patterns. Aborting BCE.
16                     f ← (C, l)
17     D ← empty deactivation index. Q ← empty priority queue.
18     if f is defined and is (C, _). then
19         Add all pairs (C, l) with l ∈ C and l is a pattern to Q.
20         Mark all pairs (C′, l′) with C′ ≠ C in φ as deactivated by C.
21     else
22         foreach (C, l) ∈ φ do
23             p ← polarity of l. h ← head of l.
24             r ← N((p, l)) if l is rigid, N((p, l)) ∪ N((¬p, l)) otherwise.
25             if r = ∅ then
26                 Enqueue (C, l).
27             else
28                 Mark (C, l) as deactivated by the clauses in r.
```

Listing 8: Initializing the data structures for pattern blocked clause elimination.

```
28
29  B ← ∅
30  while Q not empty do
31      (C, l) ← dequeue from Q.
32      S ← QueryPatternIndex(Q, l).
33      b ← 𝕋
34      foreach (C', L) ∈ S do
35          if ¬ValidOrNotRes(C,l,C',L) then
36              b ← 𝔽
37              break
38      if b = 𝕋 then
            // Clause is blocked!
39          B ← B ∪ {C}
40          Add clauses literal pairs deactivated by C to Q.
41      else
42          Mark (C, l) as deactivated by C'.
43  return M \ B
```

Listing 9: Main loop of pattern blocked clause elimination.

The last missing piece is the check for validity of the l-resolvents. Since our main loop only collects resolution candidates, the algorithm is slightly more complex. Nevertheless, it again is an adaption of the algorithm for FOL which we reproduced in listing 5. To capture the notion of l-unification constraints, we introduce an additional set P which first creates the appropriate unification problems and removes those constraints which are by themselves not solvable. Hence, after the first loop, we got rid of all unification candidates which are not actually unifiable.

```
1  function ValidOrNotRes(C, l, D, L)
       Input: A clause C, a pattern literal l ∈ C, a second clause D, and a set of
              literals L ⊆ D
       Output: 𝕋 if all l-resolvents on C with D are valid.
2      P ← ∅// For unification constraints a = b where a is l or ¬l.
3      foreach l′ ∈ L do
4          if The polarity of l is different from the polarity of l′ then
5              ρ ← (l = l′)
6          else
               // Given the overall algorithm, either l or l′ must have
                  a flex head at this point!
7              if l has a flex head then
8                  ρ ← (l = (¬l′))
9              else
10                 ρ ← ((¬l) = l′)
11         if ρ is solvable (the pair is unifiable) then
12             P ← P ∪ {ρ}
13     foreach ρ ∈ P do
14         T ← {ρ}
15         while T is solvable with MGU σ do
16             N′ ← the literals from L which are present in the constraints in T.
17             K ← (C − l)[σ] ∨ (D \ N′)[σ]
18             C ← {{t, t′} | t = t̄, t ∈ K, t′ ∈ K}
19             if C ≠ ∅ then                                   // C is valid.
20                 if for all p ∈ C there is a n ∈ L such that n[σ] ∈ p or (¬n)[σ] ∈ p
                   then
21                     T ← T ∪ {n | n ∈ L, ∃p ∈ C s.t. n[σ] ∈ p or (¬n)[σ] ∈ p}
22                 else
23                     break The while loop.
24             else
25                 return 𝔽
26     return 𝕋
```

Listing 10: Check if all l-resolvents for a clause pair are valid.

4.5 First-order Re-encoding

The finale preprocessing technique we discuss is not an adaption of any QBF or DQBF preprocessing technique. Instead, it adapts the encoding of QBFs into FOL to delay costly primitive substitution operations. Such an encoding was presented by Seidl, Lonsing, and Biere (2012) together with a tool which performs this encoding.

In fact, the tool translates input QBF problems into Effectively Propositional Logic (EPR). The target logic is a decidable fragment of classical FOL and while QBF is PSPACE-complete, deciding EPR formulas is NEXPTIME-complete. Hence, EPR is more expressive than QBF and as expressive as DQBF.

A FOL formula is in EPR if it has the following form:

$$\exists x_1. \ldots \exists x_k. \forall y_1. \ldots \forall y_l. \bigwedge_{i=0}^{n} \bigvee_{j=0}^{m_i} t_{ij}$$

where the literals t_{ij} contain no function symbols.

The translation into EPR is done in two steps. We are interested in the first step which translates QBF formulas into first-order predicate logic. The second step then would modify the result of the first step to ensure that the resulting problem has the structure as described above.

A translation function $[\![.]\!]_p$ is used to convert QBF formulas to FOL formulas:

$$\begin{aligned} [\![\exists \mathtt{x}_i. \phi]\!]_p &= \exists x_i. [\![\phi]\!]_p & [\![\forall \mathtt{x}_i. \phi]\!]_p &= \forall x_i. [\![\phi]\!]_p \\ [\![\phi \vee \psi]\!]_p &= [\![\phi]\!]_p \vee [\![\psi]\!]_p & [\![\phi \wedge \psi]\!]_p &= [\![\phi]\!]_p \wedge [\![\psi]\!]_p \\ [\![\mathtt{x}_i]\!]_p &= p(x_i) & [\![\neg \mathtt{x}_i]\!]_p &= \neg p(x_i) \end{aligned}$$

With p being a new predicate symbol. Furthermore, two dedicated unary function symbols (constants) $\top$ and $\bot$ are added. This leads to the lemma:

Lemma 4.5.1. Let ϕ be a QBF and p a unary predicate symbol. Then ϕ is satisfiable if and only if the FOL formula $[\![\phi]\!]_p \wedge p(\top) \wedge \neg p(\bot)$ is (see Seidl, Lonsing, and Biere 2012).

Intuitively, the predicate p is used to capture the behavior of QBF variables. This is expressed by the two additional unit clauses. In some sense, it encodes the properties of the Boolean type. Since HOL natively supports Boolean variables, this additional wrapping is not needed when translating from QBF to HOL.

Recall the description of primitive substitution on page 2.3.1. In the proof calculi used by Leo-II and Leo-III primitive substitution is only applied to the top-level. Hence, when directly translating QBF to HOL primitive substitution will guess Boolean assignments to variables and will guess functions for the generated Skolem functions.

Our procedure wraps literals into a fresh predicate p, just as the translation function $[\![.]\!]_p$ does. The procedure is parameterized by a Boolean value e which indicates whether equalities should be wrapped into the new predicate too. Listing 11 contains the pseudo code for the procedure. It first iterates over the matrix and wraps the literals and then adds two unit clauses $\{[p(\top)]^{tt}\}$ and $\{[p(\bot)]^{ff}\}$[6].

```
1  function FirstOrderReEncoding(M, e)
       Input: A clausified HOL problem M and a Boolean value e.
       Output: A modified matrix.
2      M' ← ∅
3      p ← a fresh function symbol of type o → o.
4      foreach C ∈ M do
5          C' ← ∅
6          foreach l ∈ C do
7              if l is of the form [t_o]^α then // l is not an equalization
               literal.
8                  C' ← C' ∪ {[(p t_o)]^α}
9              else
                   // l is of the form [s_τ = t_τ]^α.
10                 if e = 𝕋 then
11                     C' ← C' ∪ {[p (s_τ = t_τ)]^α}
12                 else
13                     C' ← C' ∪ {l}
14         M' ← M' ∪ {C'}
15     M' ← M' ∪ {{[p(⊤)]^tt}, {[p(⊥)]^ff}}
16     return M'
```

Listing 11: First-order Re-Encoding.

Given that the fresh symbol $p_{o\to o}$ is forced by the two additional unit clauses

6 The two symbols $\top$ and $\bot$ are commonly used in HOL to represent truth constants. They are defined as $\|\top\|^{\mathcal{M},\sigma} = \mathbb{T}$ and $\|\bot\|^{\mathcal{M},\sigma} = \mathbb{F}$ for an interpretations $\mathcal{M}$ and variable assignments σ.

to behave exactly like the identity function $\lambda x_o.\, x_o$, it is trivial to see, that this procedure does not change the satisfiability status of the input problem.

Resolving with the two added clauses also removes the wrapping after one step. Consider the following example:

$$\dfrac{\dfrac{\dfrac{[p(x_o)]^{tt} \qquad [p(\bot)]^{ff}}{[p(x_o) = p(\bot)]^{ff}}\,\mathcal{R}(Res)}{[x_o = \bot]^{ff}}\,\mathcal{R}(DeComp)}{[x_o]^{tt}}$$

Resolving with the unit clause creates a unification constraint between two application of the $p_{o\to o}$ function. Applying one of the pre-unification rules results in an equality between the wrapped term and a constant, which after some simplifications steps leads to the unwrapped term.

5 Implementation

We implemented the preprocessing techniques proposed in chapter 4 in the Leo-III system. In this chapter, we discuss various practical aspects related to our implementation. In section 5.1 we start the discussion with an outline of the questions which arose while implementing the techniques and our answers to these questions. To generate suitable benchmarking problems we developed tools to convert QBF and DQBF problems to HOL. A description and manual of those tools can be found in section 5.2. Finally, section 5.3 describes the benchmarking results.

5.1 Aspects of the Implementation

Overall our implementation of the preprocessing algorithms is very similar to the pseudo code given in chapter 4. Therefore, we will not reproduce the final source code here, but instead discuss various aspects of the implementation.

As mentioned in subsection 2.3.2, Leo-III is implemented in the Scala programming language. The Leo-III homepage[1] provides information and downloads[2] related to the project. The documentation included with Leo-III provides an introduction how the prover can be compiled on a Linux system.

Leo-III collects the various available rules in a *Control* object. We opted to implement our preprocessing technique as such rules. Each rule is implemented as a Scala object which implements the `CalculusRule` trait to ensures that the rule can be present in the generated proof output. The trait is simple:

```
1 trait CalculusRule {
2     def name: String
3     def inferenceStatus: SuccessSZS
4 }
```

The `name` attribute is a human readable name of the rule and the `inferenceStatus` attribute represents the inference status of the rule. Its values are taken

1 `http://page.mi.fu-berlin.de/lex/leo3/`
2 At the time of writing, version 1.1 of Leo-III was publicly available. This version did not include our additions.

from the szs ontology[3]. The szs ontology by Sutcliffe, Zimmer, and Schulz (2003) provides a systematic collection of status values for logical data and relationships between the values. For our rules the `SZS_EquiSatisfiable` status is used.

Finally, each rule object must have a method which is used by the control object to apply the rule. This is not abstracted into an interface and differs from rule to rule. In our case we implemented one method per technique which takes a set of causes as input and returns a set of clauses. In the case of universal reduction, this set contains all the original clauses after removing the removable literals. Constant extraction, on the other hand, returns a new set of unit clauses which must be added to the clause set. Blocked clause elimination returns a subset of the input clauses and first-order re-encoding also returns the input set, where some clauses are modified and two additional clauses have been added.

The preprocessing rules can be activated and parameterized by command line arguments handed to Leo-III. The arguments are:

- `ure_activate` to activate universal reduction.
- `sce_activate` to activate SAT based constant extraction.
- `bce_activate` to activate blocked clause elimination.
- `fre_activate` to activate first-order re-encoding and `fre_dontWrapEqs` to deactivate wrapping of equality literals.

Since SAT based constant extraction might produce unit clauses consisting of a singular universally quantified variable which universal reduction can than eliminate to generate the empty clause, we run SAT based constant extraction before universal reduction. Furthermore, blocked clause elimination might remove the clauses introduced by SAT based constant extraction and hence must be run first. Overall, the techniques are executed during the preprocessing phase of Leo-III in the following order:

1. Blocked clause elimination,
2. SAT based constant extraction,
3. Universal reduction,
4. First-order re-encoding.

3 A list of potential szs states is available online at `http://www.cs.miami.edu/~tptp/cgi-bin/SeeTPTP?Category=Documents&File=SZSOntology`

Executing those «complex» preprocessing methods is done after all the other preprocessing steps, especially clausification and skolemization are completed and right before the main reasoning loop starts.

Implementing universal reduction and first-order re-encoding was straightforward and our implementation corresponds to the pseudo code. Blocked clause elimination and SAT based constant extraction is a bit more complex.

To implement blocked clause elimination, we developed a `PatternIndex` class which implements the pattern index data structure as described in subsection 4.4.2. Furthermore, a class `Deactivations` is used to keep track of deactivated clauses. Internally it used two hash maps. One map is used to map clause literal pairs to the clauses that deactivate them and one is ued to map clauses to the clause literal pairs that are deactivated by them.

Our implementation of SAT based constant extraction uses an ad-hoc graph implementation to represent the equality graph. Edges are represented in an adjacency lists. Since edges are only used when making the graph chordal and during enumeration of the transitivity constraints, this representation works well. Furthermore, SAT based constant extraction utilizes the external SAT solver *pico*SAT.

5.1.1 Bindings for PicoSAT

The *pico*SAT system by Biere (2008) is a small SAT solver, which can be used as a standalone tool, or as a library. The latest release of *pico*SAT is available on the *pico*SAT homepage[4].

While *pico*SAT is not competitive with recent SAT solvers[5], its well documented interface and decent performance made it a good choice for the implementation of SAT based constant extraction. Furthermore, the system supports two important features. On one hand, iterative solving is supported. This means, that additional clauses can be added as assumptions after running the solver without invalidating the old intermediate data structures. In the case of SAT based constant extraction, the individual literals can be temporarily assumed. On the other hand, *pico*SATalso supports the generation of *unsatisfiability cores*. In the case of an unsatisfiability problem, this feature allows the generation of subsection of clauses which are unsatisfiable by themselves. For SAT based constant extraction, this allows the detection which original clauses imply the constant literal. Note that the generated cores are not minimal. Hence,

4 `http://fmv.jku.at/picosat/`

5 For example, *pico*SAT did not participate in the SAT competition 2016. Results are available here: `http://baldur.iti.kit.edu/sat-competition-2016/index.php?cat=results`

there might be a real subset of clauses which is still unsatisfiable. The PicoSAT system comes with a standalone tool to compute minimal unsatisfiability cores, this feature, however, is not present in the API, but is also not necessary for our implementation.

The API of PicoSAT is completely defined in a single header file `picosat.h`. Every function in this file is commented and this also serves as the main documentation of the PicoSAT API. To facilitate the usage of PicoSAT in Leo-III, we implemented Java Native Interface (JNI) bindings. This allows calls to PicoSAT directly from Leo-III without invoking PicoSAT as an external command. A documentation for JNI is available on the Java web page[6].

Our bindings mirror the functions of the `picosat.h` file and only provide minimal abstraction around them. The PicoSAT API allows the creation of multiple solver instances which can be used independently. We represent this in our API too. It is unclear if PicoSAT is thread save and we suggest limiting the usage of each PicoSAT solver instance to one thread. Furthermore, unsatisfiability core generation has a bigger memory footprint, since prove traces must be stored. Therefore, our API allows the deactivation of unsatisfiability core generation for each solver instance.

The following code gives a short example of the API:

```
1 val solver = PicoSAT(true)
2 solver.addClause(List(-1, -3, -2))
3 solver.addClause(List(-2, -3, -1))
4 solver.addClause(List(3))
5 assert(solver.solve() == PicoSAT.SAT)
6 assert(context.getAssignment(1) == Some(false))
```

In the first line a new solver instance with activated core generation is requested. Then three clauses are added. Literals are expressed as integers. A positive integer n stands for the propositional literal $\mathtt{x}_n$ and a negative integer $-n$ for $\neg\mathtt{x}_n$. Finally, the solver is called and returns that satisfiability state of the problem. Now the generated model of the problem can be queried. In this case $\mathbb{F}$ must be assigned to $\mathtt{x}_1$.

5.2 From QBF to HOL: QBFToys

To appropriately benchmark the preprocessing techniques we can use the problems present in the TPTP problem library. We, however, based our preprocessing techniques on QBF preprocessing and hence hoped evaluating their

6 http://docs.oracle.com/javase/8/docs/technotes/guides/jni/index.html

performance on QBF and DQBF problems would give valuable insights. Unfortunately the problems we generated this way were too hard for Leo-III and ultimately unusable. See subsection 5.3.1 for a discussion of this problem.

We implemented tools which perform the translations from QBF and DQBF to HOL as described in section 3.1 and 3.3. The tools are available on GitHub[7].

Building the Tools

Both tools are written in Haskell. While there are multiple ways to build the tools, using the *Stack* build tool[8] is strongly recommended by us and documented here. To install Stack follow the instructions on the Stack homepage.

The `qbfTool` folder contains the source code for the QBF to HOL converter, and the DQBF tool is in the `dqbfTool` folder. After the installation of Stack is complete, open a terminal and navigate to either folder. Then run then following commands, where `{tool}` is either `qbfTool`, or `dqbfTool`:

```
1 > stack setup
2 > stack build
3 > stack exec {tool}
```

The `stack build` command outputs the path the resulting binary is placed in. On Linux calling `stack install` will copy the binary to `~/.local/bin`. Alternatively, the programs can be run with additional arguments (`{args}`) by calling

```
1 > stack exec {tool} -- {args}
```

Converting QBF Problems

The QDIMACS format is widely used to represent QBF problems in CNF. It is an extension of the DIMACS format used for propositional problems and QDIMACS is compatible with DIMACS. Every valid DIMACS file is also a valid QDIMACS file. Variables in the QDIMACS format are expressed by non-zero natural numbers and literals are expressed by non-zero integers. The variable index of a literal is the absolute value of the integer, the polarity is represented by the sign of the integer. The following is a small QDIMACS file:

```
1 p cnf 3 3
2 c an example problem
```

7 https://github.com/hansjoergschurr/QBFToys
8 Available at: https://haskellstack.org

```
3 a 1 2
4 e 3
5 -1 -3 -2 0
6 -2 -3 -1 0
7 3 0
```

The first line starting with `p` gives meta information about the problem. The first number is the highest variable appearing in the problem and the second number denotes the number of clauses. Then lists of literals are given line by line and terminated by a `0`. The first few lines represent the quantifier prefix and start with a letter. The letter `e` is used for existential quantification and `a` for universal quantification. Then the clauses are given one per line. Comment lines start with `c`.

A formal grammar of the QDIMACS format can be found online[9]. This grammar forbids empty clauses and files without any clauses. We found that BLOQQER sometimes does generate problems with empty clauses and completely empty problems. Hence, our tool tries to be very liberal when parsing QDIMACS problems. It especially ignores the metadata given in the first line.

The QBF2EPR tool by Seidl, Lonsing, and Biere (2012), which translates QBFS to EPR also allows the translation into HOL and supports TPTP output. This feature of the tool, however, is implemented in Python and is too slow on bigger input problems. Furthermore, input problems which did not follow the QDIMACS standard result in crashes.

The default behavior of `qbfTool` is to accept a QDIMACS problem on the standard input and to output a TPTP problem to standard output. Assuming a file `test.qdimacs` which contains the following QDIMACS problem:

```
1 p cnf 6 3
2 e 1 0
3 a 0
4 e 2 0
5 a 3 4 0
6 e 5 6 0
7 1 3 5 0
8 1 2 0
9 2 4 6 0
10 99 0
```

This problem is not a valid QDIMACS file. The meta values are wrong and a quantifier line contains an empty list of variables. Executing `qbfTool` on this

9 `http://www.qbflib.org/qdimacs.html`

problem results in the following output:

```
1 > cat test.qdimacs | stack exec qbfTool
2 thf(c,conjecture,(? [X1: $o,X2: $o,X3: $o]:(! [X4: $o,X5:
  ↪ $o]:(? [X6: $o,X7:
  ↪ $o]:((X1|X4|X6)&(X1|X3)&(X3|X5|X7)&(X2))))))).
```

This output is not a direct translation of the input problem. The `qbfTool` also applies some normalization steps. Those are:

- Consequitive lines containing the same quantifier are merged.
- Empty quantifiers are removed and the surrounding quantifiers are merged.
- Variables are renamed by the order they appear in the quantifier. This also assures that all variables are used without gap.
- Variables which are used in the matrix, but which do not appear in the quantifier prefix, are added to the outermost quantifier if the outermost quantifier is an existential quantifier. Otherwise, an existential quantifier containing those variables is added.
- If the input problem does not contain any clauses, the clause $x \vee \neg x$ is added.
- If the input problem contains the empty clause, a clause x and a clause $\neg x$ is added.

If the tool is called with the command line parameter `-n`, the output is a valid QDIMACS problem resulting from these transformations instead of a TPTP problem. For the QDIMACS problem given above this output is:

```
1 p cnf 7 4
2 e 1 2 3 0
3 a 4 5 0
4 e 6 7 0
5 1 4 6 0
6 1 3 0
7 3 5 7 0
8 2 0
```

As described above, free variables are *implicitly* existentially quantified in the input problem and *explicitly* existentially quantified in the output problem. Hence, the input QBF problem is satisfiable if the output problem is valid. A resolution prover will negate the input problem and show its unsatisfiability. Negating a problem in clause normal form and clausifying the negated problem requires repeated application of the distributive laws and is therefore an expensive operation. Furthermore, a prover is geared towards showing validity and will seldom deduce that an input problem is not valid. Hence, the `-i` command line parameter will instruct `qbfTool` to add a negation in front of the output problem. The resulting problem will be a theorem if an only if the input problem is unsatisfiable.

Finally, the `-o filename` parameter can be used to instruct `qbfTool` to write the output into a file. If this parameter is used, `qbfTool` will output exactly one line to the standard output. This line will contain the number of variables and number of clause in the problem separated by a space.

Converting DQBF Problems

The `dqbfTool` generates HOL problems in the TPTP format from DQBF problems in the *bunsat* format. This format has been defined for the *bunsat* tool by Finkbeiner and Tentrup (2014) which is an incomplete unsatisfiability checker for DQBF. This format also supports input problems which are not in CNF.

As an alternative to the *bunsat* format, the DQDIMACS format exists. This format is an extension of the QDIMACS format and was developed by Fröhlich et al. (2014) for the *i*DQ system. Like QDIMACS every valid (Q)DIMACS file is also a valid DQDIMACS file. Unlike the *bunsat* format DQDIMACS only supports problems in CNF.

The following example of the *bunsat* format was taken from the *bunsat* web page[10]. This page also contains an incomplete grammar of the format. We used the source code of *bunsat* tools published on the web page to «reverse engineer» the missing pieces of the grammar.

```
1 A x1 x2: E{x1} y1 : E{x2} y2: ((~y1) | y2) <--> (x1 ^ x2)
```

Problems in *bunsat* formats start with the universal quantifier denoted by an `A`, follow with the list of universal variables. Then, after a colon, follows a list of existentially quantified variables each of the form `E{x1 ... xn} yi` where the variables in the curly brackets are the dependency set of the variable `yi`. Then, after another colon, follows a Boolean formula. The format supports

10 https://www.react.uni-saarland.de/tools/bunsat/

multiple synonymous names for the operators. To express conjunction the `&` operator can be used and `^` denotes the exclusive-or.

The `dqbfTool` behaves the same way as the `qbfTool` behaves, but does not apply any normalizations to the input problem. Hence, the `-n` command line parameter is not supported. The `-i` parameter adds a negation symbol to the output and the `-o` parameter writes the output into a file. If the `-o` parameter is given, the tool does not write any metadata to the standard output.

5.3 Benchmarking

We now discuss the results of benchmarking our preprocessing techniques on various problems. This section starts with a description of the benchmarking problems we intended to use. In accordance to the previously discussed logics, this set of problems is threefold. We prepared QBF, DQBF, and HOL problems for benchmarking.

The QBF dataset is based on the *eval2010* set of problems. This set of QBF problems was used for the QBFEVAL'10 competition[11]. It consists of problems selected from the QBF*lib* database[12] of QBF problems and additional contributed problems. The dataset[13] contains a total of 568 problems in the QDIMACS format. Some of these problems are satisfiable, some of them are not. Before translating the problems to HOL problems, we classified them using the D*ep*QBF QBF solver. Problems which could not be solved by D*ep*QBF were discarded, there is not much hope that problems to hard for a specialized QBF solver can be solved by a HOL theorem prover. Overall this process[14] resulted in 116 satisfiable problems and 201 unsatisfiable problems. Hence, 317 problems were selected. In a second step we converted all problems to HOL problems in the TPTP format. Since the `qbfTool` adds explicit existential quantification, the unsatisfiable problems were negated. Hence, all resulting problems are theorems[15].

Furthermore, we downloaded the DQBF benchmarks published by the *bunsat* authors[16]. According to Finkbeiner and Tentrup (2014), these problems have

11 see: `http://www.qbflib.org/event_page.php?year=2010`

12 The library is available online at `http://www.qbflib.org/`.

13 The version used for our benchmarks were downloaded from the web page of the QBFGALLERY 2013. This was a non-competitive evaluation of QBF systems held in 2013 and 2014. The results and benchmark downloads, including the used QBFEVAL'10 problem set, are available at `http://www.kr.tuwien.ac.at/events/qbfgallery2013/results.html`.

14 We used D*ep*QBF version 6.02 in its default configuration with a timeout of 30 seconds on a shared computer with two Intel Xeon E5430 quad core CPUs and 32GB RAM.

15 Assuming that D*ep*QBF and our conversation tool is sound.

16 available at: `https://www.react.uni-saarland.de/tools/bunsat/`

been generated by taking a Boolean circuit, replacing parts of the circuit with «black boxes» and then taking the original circuit as specifications for the resulting partial circuit. The so generated DQBF problem is satisfiable if the «black boxes» can be filled with Boolean circuits to construct a circuit with the same behavior as the specification circuit. Since *bunsat* is an *unsatisfiability* checker, exactly one random gate was replaced with another gate to make the problem unsatisfiable. Hence, when converting the problems with the `dqbfTool`, we negated the problem to generate a theorem. The input dataset contains problems generated from four circuits: A 32-bit adder, a 32-bit lookahead arbiter implementation, a 32-bit multiplexer, and a 4-bit multiplier. Into this circuit one, three, five, seven, or nine black boxes were inserted. For each of this twenty classes, 100 problems are part of the problem set. We sampled 25 problems from each class to get 500 DQBF problems in total.

Finally, we selected the set of TPTP problems which were eligible for the CASC-J8 HOL track[17]. This is a set of 743 HOL problems from the TPTP library. Unfortunately, as we describe in the next section, we encountered difficulties with the QBF and DQBF problem sets during the empirical evaluation and were ultimately unable to use it.

5.3.1 Empirical Results

We ran various empirical experiments on the above described problem sets. The experiments were run on a shared Debian Linux server with two Intel Xeon E5430 quad core CPUs and 32GB RAM running version 3.6. of the Linux kernel.

We used a Python script to automatize the tests. The script consumes a list of input problems and runs a copy of Leo-III on each problem one after another. It also matches the log output of Leo-III against multiple regular expressions which are used to extract values from the log. For example the expression «`^\% URE Time (\d+(.\d*)?)`» extracts a decimal value from the log. In this case an expression was inserted into Leo-III source code which printed the runtime of universal reduction. Furthermore, if performance parameters were measured, Leo-III was forced to abort as soon as the parameter in question was calculated. This means, that if a value could not be measured because of a timeout, preprocessing was not even finished.

For these experiments our preprocessing techniques were added to Leo-III version 1.1. All experiments were run with a timeout of 60 seconds. Our

17 The list of eligible problems is available at `http://www.cs.miami.edu/~tptp/CASC/J8/EligibleProblems.html`.

	Problem Name	Variables	Clauses
sat	toilet_g_04_01.2-shuffled	22	52
	toilet_g_02_01.2-shuffled	12	22
unsat	lut4_XOR_fOR-shuffled	43	168
	toilet_c_04_01.4-shuffled	58	185
	z4ml.blif_0.10_1.00_0_0_out_exact-shuffled	63	194
	toilet_c_02_01.2-shuffled	17	37
	toilet_a_02_01.2-shuffled	18	39

Table 5.1: Feasible QBF problems.

script waits 30 seconds longer before forcibly killing Leo-III. Since the internal timeout procedure of Leo-III did not work reliably before the prover finished its preprocessing part, this often resulted in an effective timeout of 90 seconds. Leo-III was run in sequential mode.

The Infeasibility of Some Problems

During our first experiments with the problem sets generated by translating QBF and DQBF problems we encountered a challenge: In many instances the prover would not even reach the point where our preprocessing procedures are executed before the timeout is reached. This means that parsing, clausification, and build-in preprocessing techniques already took longer then the set timeout.

We decided to not use the DQBF problem set for further experiments and to select those problems from the QBF problem set, where the internal preprocessing steps where finished before the timeout.

The result of this test showed how hard this problem domain is for HOL theorem provers. Of the 317 problems, only seven problems passed minimal processing before the timeout was reached. Two problems were satisfiable, and five problems were counter satisfiable. Table 5.1 shows the remaining problems and the number of distinct variables and clauses appearing in those problems.

We included these seven problems into every one of the following experiments. This resulted in 750 problems in total.

The experiments were conducted in two rounds. In the first round, we applied the algorithms to all input problems and recorded the number of problems Leo-III was able to solve. In the second round, we measured several key

metrics, such as the runtime, for each algorithm. Leo-III was aborted after recording these measurements.

Impact on the Number of Solved Problems

To gauge the impact of the preprocessing techniques on the number of problems Leo-III is able to solve, we ran the prover on the dataset described above with multiple configurations.

Of the 750 theorems, Leo-III was able to prove

- 475 without preprocessing (None),
- 475 with universal reduction (URE),
- 477 with SAT based constant extraction (SCE),
- 454 with blocked clause elimination (BCE),
- 418 with first-order re-encoding (Fo-Re), and
- 458 with all techniques except first-order re-encoding (Mult.).

The shorthand noted in brackets is used in the following tables to reference the configurations. The last configuration was selected after preliminary experiments showed that first-order re-encoding has a negative impact on prover performance.

Overall, only SAT based constant extraction enabled the prover to solve more problems. Each algorithm, however, made some problems unsolvable, and some other problems solvable. Hence, we calculated the number of problems which were solved by one configuration compared to another one.

In table 5.2 each number denotes the number of problems which were solved by the configuration in the column, but not by the configuration in the row. For example, Leo-III with universal reduction was able to solve seven problems which were not solved by Leo-III with SAT based constant extraction.

This shows, that every technique was able to solve some problems which were not solved by Leo-III without any additional preprocessing. Furthermore, each technique was able to solve appropriately 60 more problems than Leo-III with first-order re-encoding. Hence, this technique is not very promising.

Only one problem (`SEU717^1`[18]) was solved by the configuration *Mult.* and none of the three specific preprocessing techniques alone, nor by Leo-III with-

18 This problem is from the set theory domain and contains the laws of typed sets. It is available at `http://www.cs.miami.edu/~tptp/cgi-bin/SeeTPTP?Category=Problems&Domain=SEU&File=SEU717^1.p`

		Solved					
		None	Mult.	URE	Fo-Re	SCE	BCE
Not Solved	None	0	11	7	7	10	7
	Mult.	28	0	24	25	27	9
	URE	7	7	0	7	9	8
	Fo-Re	64	65	64	0	65	58
	SCE	8	8	7	6	0	7
	BCE	28	13	29	22	30	0

Table 5.2: Problems solved by various preprocessing techniques.

out preprocessing. Since the success on this singular problem might be the result of fluctuations in available computing resources, this result indicates that the techniques do not profit from one another.

Universal Reduction

In the case of universal reduction, we measured the runtime of the algorithm and the number of removed literals.

The runtime was collected for 739 problems. Hence, for 11 problems preprocessing was not finished before the timeout was reached. The mean of the runtime was 32.71 ms, the minimal runtime was 7.9 ms and the maximal runtime was 713.88 ms. Figure 5.1 shows the runtime distribution of universal reduction relative to the number of clauses arriving on the algorithm. Since Leo-III performs some processing during clausification, this number might not correspond to the number of clauses in the input problem. Overall, the performance of this algorithm is acceptable.

Universal reduction was able to remove literals from 184 problems. This includes the two satisfiable QBF problems from which universal reduction could remove 4201[19] and 50[20] literals. On the HOL problems, universal reduction could remove between one and seven literals. However, those problems were all in the domain of set theory (SEU) and software verification (SWW).

Overall, this together with table 5.2 shows that universal reduction is only useful for certain instances.

19 from the `toilet_g_04_01.2-shuffled` problem.
20 from the `toilet_g_02_01.2-shuffled` problem.

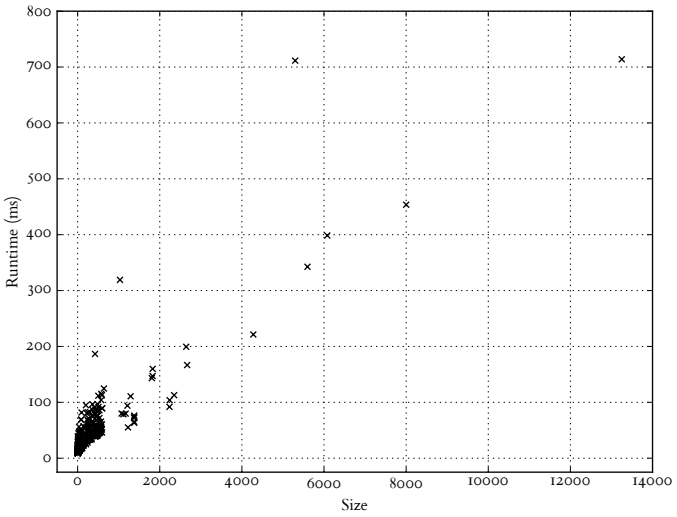

Figure 5.1: Runtime of universal reduction.

SAT Based Constant Extraction

As metrics for SAT based constant extraction we used the number of propositional clauses and literals, as reported by *pico*SAT, both before adding transitivity constraints and after. We also saved the number of SAT calls and, naturally, the number of added unit clauses.

To compare this algorithm with the other preprocessing algorithms, we calculated the runtime relative to the number of clauses. Figure 5.2 is a scatter plot of the runtime. Overall, the algorithm ran between 102.3 ms and 2231.02 ms with a mean runtime of 232.78 ms and a median runtime of 149.33 ms. The SAT solver was called 40.79 times on average, but because of a single problem with 1250 calls, the median number is only eleven calls.

Unit clauses were added to 76 problems distributed over 8 of the 25 HOL domains and to the QBF problems. For 38 problems one unit clause was added, two unit clauses were added to 7 problems, and three unit clauses were added to 18 problems. 13 problems were augmented with more than three unit clauses.

Additional transitivity constraints were added to 248 problems. In the case of 58 of these problems the algorithm was able to add also add unit clauses to the prolem.

As the comparison with the other algorithms indicates, SAT based constant extraction is an effective algorithm. Nevertheless, it still works only for certain instance domains, and is slower than the other algorithms. Furthermore, the

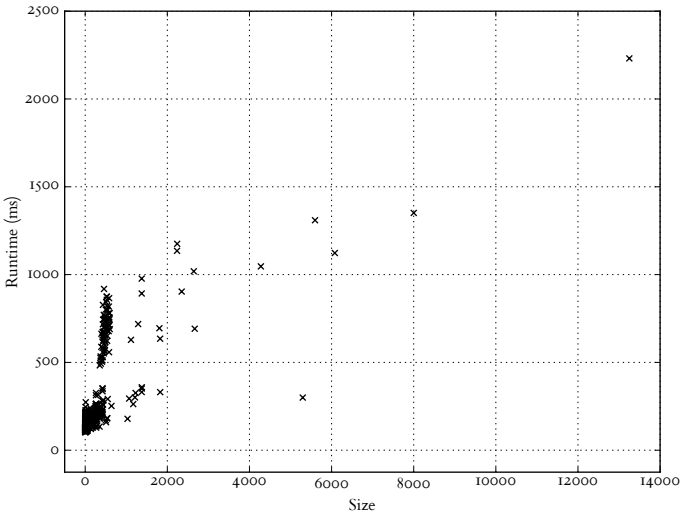

Figure 5.2: Runtime of SAT based constant extraction.

addition of transitivity constraints seems to be a valuable addition.

Pattern Blocked Clause Elimination

In the case of Pattern Blocked Clause Elimination, we marked problems which were not equational, problems where two non-pattern literals with flex head appeared, and problems with one non-pattern literal with flex head. In the former two cases, pattern blocked clause elimination can not be applied. Furthermore, we also saved the number of clauses removed by this procedure.

Overall, 261 problems were not equational. Of those 261 problems, 124 contained two clauses with flex headed non-patterns. The remaining 137 problems were candidates for blocked clause elimination.

One problem resulted in an extremely long runtime of 47.5 seconds. This problem was one of the unsatisfiable QBF problems[21]. The negation of this problem resulted in long clauses, which in turn inflated the runtime of the blocked clause elimination algorithm. Because of this outlier the average runtime was 201.7 ms and the median runtime was 25 ms with a minimal runtime of 18.84 ms. This includes equational problems. After removing the equational problems, the median runtime is 31.1 ms. Figure 5.3 shows the runtime of blocked clause elimination and omits the problem with over 40 seconds

21 the `lut4_XOR_fOR-shuffled` problem.

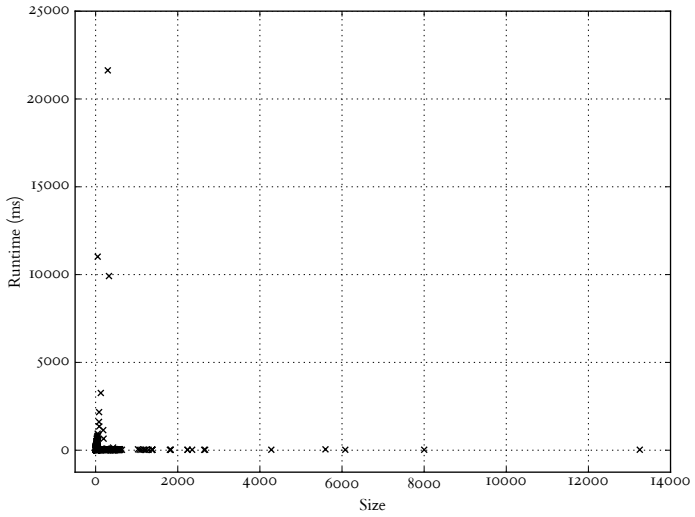

Figure 5.3: Runtime of pattern blocked clause elimination.

runtime.

The algorithm was able to remove clauses from 37 input problems. For ten of those problems, only one clause could be removed. For the problem SYO377^5[22] the algorithm could remove 42 clauses.

Blocked clause elimination is seriously constrained by the tight requirements on the input problems. In those cases where the algorithm is applicable, however, it was able to remove at least one clause in 27% of the cases.

First-order Re-encoding

Since first-order re-encoding is done during one iteration over the problem, the only meaningful metric is the algorithm runtime. The runtime is expected do be linear in the total number of literals appearing in the input problem.

Figure 5.4 shows a scatter plot of the runtime. The linear behavior is visible. Overall, the runtime varied between 4.88 ms and 389.68 ms, with a median runtime of 9.78 ms. The mean runtime was 16.23 ms.

As we have already seen, applying first-order re-encoding was detrimental for the solvability of many problems. Nevertheless, table 5.2 indicates that seven problems were solved when first-order re-encoding was applied and not

22 This problem from the syntactic domain is available at `http://www.cs.miami.edu/~tptp/cgi-bin/SeeTPTP?Category=Problems&Domain=SYO&File=SYO377^5.p`

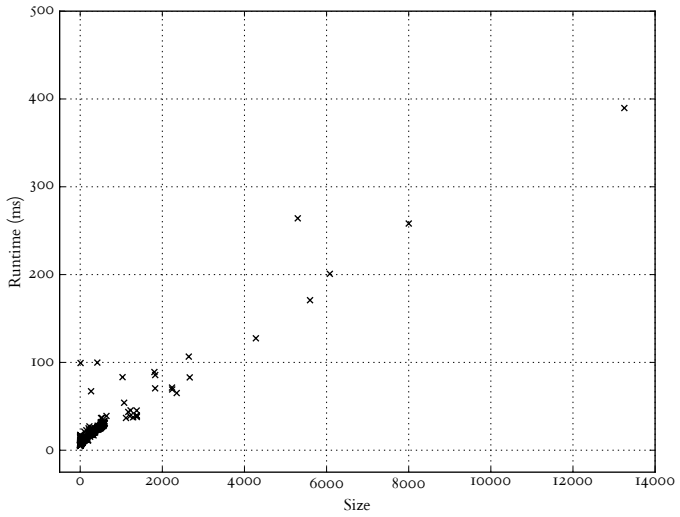

Figure 5.4: Runtime of first-order re-encoding.

by the pure Leo-III system. Of those seven problems, three were not solved by any other configuration. Those three problems were `LCL727^5`, `AGT031^1`, and `SYO213^5`. Since these problems are from different domains and do not have any obvious similarities, we suspect that this coincidentally and not the result of applying first-order re-encoding.

Conclusion

As we have seen, only SAT based constant extraction is clearly beneficial to the proof search. Pattern blocked clause elimination, in its current iteration, and universal reduction are only useful in some cases. There is no empirical evidence that first-order re-encoding is useful at all. Fortunately, all four algorithms have a useful runtime behavior.

Leo-III is a resolution based prover. This means, that it tries to find a proof by creating new clauses. Blocked clause elimination and universal reduction are not necessary helpful during this process, since they might remove elements which are useful during this search process. Constant extraction, on the other hand, aids this process by adding new, short clauses.

6 Insights

This chapter concludes this thesis. In the previous chapters we introduced HOL and QBF, discussed preprocessing techniques for QBF, described the adaption to HOL of some of those preprocessing techniques, and the practical implementation of them. We now set this work into a wider context. To do so, we present some related work in section 6.1 and suggest some further work in section 6.2. Section 6.3 concludes the thesis with some final remarks.

6.1 Related Work

Many of the preprocessing techniques for QBF solving were originally developed for SAT solving. A tool for SAT preprocessing has been SATELITE (Eén and Biere 2005). Among other techniques, it implemented variable elimination by self-subsuming resolution. Since its publication, it has been subsumed by recent releases of the SAT solver MINISAT2[1]. Another technique used for preprocessing is hyper binary resolution (Bacchus and Winter 2004). Blocked clause elimination was first introduced for SAT by Kullmann (1999) (cf. Järvisalo, Biere, and Heule 2010).

Preprocessing has received some attention by state of the art theorem provers for first-order logic. We already discussed the recent application of blocked clause elimination to FOL. Earlier work in this area was done by Hoder et al. (2012). This work was motivated by industrial applications of first-order reasoning. The authors investigate two kinds of techniques. On one hand, they present methods for eliminating and simplifying definitions that also result in a clausification process which preserves the EPR fragment. On the other hand, they present a generalization of the And-Inverter Graph (AIG) datastucture, which they call Quantified And-Invert Graph (QAIG). The authors then use this datastucture to adapt various preprocessing techniques for SAT to FOL.

As we discussed, QBF and DQBF reasoners are *satisfiability checkers*. Another class of problems in this domain is Satisfiability Modulo Theories (SMT). As the name suggest, SMT languages enrich propositional satisfiability checking

1 See the SATELITE homepage: `http://minisat.se/SatELite.html`

with theories such as uninterpreted functions and equality. Systems for solving SMT problems are commonly built from two parts. A SAT solver reasons on a propositional abstraction of the formulas generated by the theory reasoner. If the SAT solver finds a model for a problem, this model is analyzed by the theory reasoner. If the theory reasoner can create a model for the original problem, then the process terminates, otherwise the SAT problem is refined (cf. Barrett, Sebastiani, et al. 2009).

Examples for SMT solvers are Boolector (Niemetz, Preiner, and Biere 2014), CVC4 (Barrett, Conway, et al. 2011), veriT (Bouton et al. 2009), and Z3 (Moura and Bjørner 2008). Preprocessing is a part of SMT solving and while the solving systems incorporate preprocessing routines, the SMTpp (Bonichon et al. 2015) tool is a standalone preprocessor. Recently, Barbosa, Blanchette, and Fontaine (2017) presented a method to generate proofs for the results of SMT formula processing.

While our focus was on theorem proving for HOL, other tasks which can be automatized exist. One such task is *model finding*. Model finders try to solve the problem: Given a HOL formula, is this formula satisfiable? These tools are especially useful if used as *count model finder*. If we have a formula of unknown status, we can apply a theorem prover and also negate the formula and apply a model finder. If the theorem prover succeeds, we know that the formula is a theorem. If the model finder succeeds, we know that the problem is counter-satisfiable. An example of a model finder is Nitpick (Blanchette and Nipkow 2010).

A form of preprocessing is also part of the *Sledgehammer* tool by (Paulson and Blanchette 2010). As discussed in subsection 2.3, Sledgehammer calls automated provers to assist the proof construction in an interactive prover. To make this process feasible, Sledgehamemr must select a small subset of theorems from the big library of theorems available in interactive provers. The selected theorems are then added to the problem for the automated prover as axioms.

6.2 Further Work

In this thesis we only investigated a limited number of QBF and DQBF preprocessing algorithms. The remaining algorithms are a obvious topic to continue this work. It might be especially fruitful to investigated dependency schemata. Wimmer, Gitina, et al. (2015) describe a simple dependency schema for DQBF. The goal of dependency schemata is to determine that some universally quantified variable x in the dependency set of an existentially quantified variable y

can be omitted because it will not be used by the Skolem function of y. In such a situation the variable x can be removed from the dependency set of y. The authors give a simple sufficient criterion for the independence of two variables and note that more complex schemata have been developed for QBF.

On the more practical side of things, implementing our preprocessing techniques as a stand-alone tool would enable us to evaluate the effect of the techniques on other theorem provers and automated tools such as model finders. This is especially interesting for BCE, since Kiesl et al. (2017) report a positive impact of BCE on solving first-order non-theorem problems. To show the that a first-order formula is not a theorem, a search for a counter-model is required. Such a tool would take a HOL formula in TPTP format as input and output a new, satisfiability equivalent, formula in the same format. It should also be able to create prove traces of transformations it applies.

Another further topic is to complete the work on BCE for HOL:

Blocked Clause Elimination

Unfortunately, our discussion of BCE for HOL was very restricted. While we presented a first implementation of BCE for the automated theorem prover Leo-III and evaluated its practical feasibility.

The most apparent missing piece is certainly the lack of a proof of soundness of the BCE procedure. The proof for the redundancy of FOL blocked clauses given by Kiesl et al. (2017) relies on the Herbrands theorem. This theorem states, that an equality free FOL formula is satisfiable if and only if every finite set of ground instances is satisfiable (cf. Fitting 1996). Then in a set of ground instances of a formula F with a clause C blocked by the literal l, the truth value of l in an assignment which falsifies C can simply be flipped to satisfy C. This new assignment will still satisfy all the clauses in $F \setminus \{C\}$ which are satisfied by the old assignment. A slightly more intricate, but similar argument is used for the case of equality blocked clauses.

A version of the Herbrand theorem for HOL exists. It was given by Miller (1987) and relies on the notion of expansion-tree proofs. We believe that this can be used to prove the soundness of our variant of blocked clause elimination.

Secondly, the lack of support for equality in our variant of BCE is a very undesirable property. Again we hypothesize, that the approach by Kiesl et al. (2017) can be extended to HOL. Handling flex and rigid literals would become more complex, and so would deciding validity of flat l-resolvents. However, the practical implementation of BCE for FOL with equality already abandoned full congruence closure in favor of a greatly simplified algorithm. Hence, an incomplete algorithm for deciding the validity might be enough. Further-

more, the notion of flattening tightly corresponds to unification constraints generated during pre-unification (cf. Huet 1975).

Overall, a fine analysis of the concept of blocking literals and clauses in the context of HOL and especially resolution based provers for HOL would be a fruitful undertaking.

Bit Precise Reasoning with HOL

In hard- and software verification, reasoning on the level of bits its often relevant for the verification process. In practice QBF and DQBF formulas can be used to do this. Another formalism used in this domain is *bit vector logic*.

Those logics provide operators on vectors of bits, such as element wise disjunction and shift operators. Variables range over bit vectors of fixed width. The solving complexity depends on the chosen language features (cf. Biere 2014).

As we have seen, HOL can also be used to describe Boolean problems. Common bit vector logics can probably be expressed in HOL by choosing functions on Booleans with the appropriate arity. This would enable the users to formulate some properties in the general language of HOL. Unfortunately, our investigation into solving QBF and DQBF problems with HOL theorem provers has shown, that they do not perform very well on such problems.

A fruitful way of approaching the problem of bit precise reasoning in HOL could be to combine bit vector and QBF reasoners with HOL theorem provers. One major challenge would be the extraction of Boolean problems from the overall clause set.

6.3 Conclusion

This thesis started with an introduction into two distinct logical languages. First we described Higher-order logic, a logic which allows quantification over functions and properties. Then we presented quantified Boolean formulas, a generalization of propositional logic, which allows quantification over Boolean values. Those two logics are quite distinct. Higher-order logic is undecidable, which quantified Boolean formulas are not. On the other hand, we showed that QBFs are a fragment of HOL. We then focused on preprocessing techniques for QBF and presented adaptions of those techniques to the task of proving HOL formulas.

Overall we introduced four techniques: Universal reduction, first-order reencoding, SAT based constant extraction, and pattern blocked clause elimina-

tion without equality. Not only were they discussed theoretically, but they were also implemented for the Leo-III system. We also evaluated the implementations empirically. Finally, we discussed related work and suggested various ways to continue working on this topic.

Our work connected two worlds: The world of HOL and the world of QBF. While QBF is fragment of HOL on the theoretical level, the world of QBF turns out to be much different. Through the use of translation tools, we were able to show, that QBF and DQBF problems are very hard, even intractable, for state of the art HOL provers. Preprocessing methods for QBF, on the other hand, struggle with the expressiveness of HOL.

While our benchmarks show a promising impact of our preprocessing techniques, we believe, that this comparison is the main contribution of the diploma thesis at hand.

Bibliography

Alama, Jesse (2016). «The Lambda Calculus». *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2016. Metaphysics Research Lab, Stanford University.

Andrews, Peter Bruce (1971). «Resolution in Type Theory». *The Journal of Symbolic Logic* 36, pp. 414–432. DOI: 10.2307/2269949.

Andrews, Peter Bruce (1989). «On connections and higher-order logic». *Journal of Automated Reasoning* 5.3, pp. 257–291. DOI: 10.1007/BF00248320.

Andrews, Peter Bruce (2014). «Church's Type Theory». *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2014. Metaphysics Research Lab, Stanford University.

Bacchus, Fahiem and Jonathan Winter (2004). «Effective Preprocessing with Hyper-Resolution and Equality Reduction». *Theory and Applications of Satisfiability Testing: 6th International Conference. Selected Revised Papers*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer, pp. 341–355. DOI: 10.1007/978-3-540-24605-3_26.

Barbosa, Haniel, Jasmin Christian Blanchette, and Pascal Fontaine (2017). «Scalable Fine-Grained Proofs for Formula Processing». *CADE 26: 26th International Conference on Automated Deduction. Proceedings*. Ed. by Leonardo de Moura. Cham: Springer International Publishing, pp. 398–412. DOI: 10.1007/978-3-319-63046-5_25.

Barrett, Clark, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli (2011). «CVC4». *Computer Aided Verification: 23rd International Conference, CAV 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

Barrett, Clark, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli (2009). «Satisfiability Modulo Theories». *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn J.H. Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. Amsterdam, The Netherlands: IOS Press. Chap. 26, pp. 825–885. DOI: 10.3233/978-1-58603-929-5-825.

Benzmüller, Christoph (1999). «Extensional Higher-Order Paramodulation and RUE-Resolution». *Automated Deduction – CADE-16*. Ed. by Harald Ganzinger. Lecture Notes in Computer Science 1632. Springer, pp. 399–413. DOI: 10.1007/3-540-48660-7_39.

Benzmüller, Christoph (2002). «Comparing Approaches to Resolution based Higher-Order Theorem Proving». *Synthese* 133.1–2, pp. 203–235. DOI: 10.1023/A:1020840027781.

Benzmüller, Christoph (2015). «Higher-Order Automated Theorem Provers». *All about Proofs, Proof for All*. Ed. by David Delahaye and Bruno Woltzenlogel Paleo. Mathematical Logic and Foundations. London, UK: College Publications, pp. 171–214. ISBN: 978-1-84890-166-7.

Benzmüller, Christoph and Michael Kohlhase (1998). «LEO – A Higher-Order Theorem Prover». *Automated Deduction – CADE-15*. Ed. by Claude Kirchner and Hélène Kirchner. Lecture Notes in Computer Science 1421. Springer, pp. 139–143. DOI: 10.1007/BFb0054256.

Benzmüller, Christoph and Dale Miller (2014). «Automation of Higher-Order Logic». *Handbook of the History of Logic, Volume 9 – Computational Logic*. Ed. by Dov M. Gabbay, Jörg H. Siekmann, and John Woods. North Holland, Elsevier, pp. 215–254. DOI: 10.1016/B978-0-444-51624-4.50005-8.

Benzmüller, Christoph, Lawrence C. Paulson, Nik Sultana, and Frank Theiß (2015). «The Higher-Order Prover LEO-II». *Journal of Automated Reasoning* 55.4, pp. 389–404. DOI: 10.1007/s10817-015-9348-y.

Benzmüller, Christoph and Bruno Woltzenlogel Paleo (2015). «Higher-Order Modal Logics: Automation and Applications». *Reasoning Web 2015*. Ed. by Adrian Paschke and Wolfgang Faber. Lecture Notes in Computer Science 9203. Berlin, Germany: Springer, pp. 32–74. DOI: 10.1007/978-3-319-21768-0_2.

Bertot, Yves and Pierre Castran (2010). *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st ed. Springer Publishing Company, Incorporated. DOI: 10.1007/978-3-662-07964-5.

Biere, Armin (2008). «PicoSAT Essentials». *Journal on Satisfiability, Boolean Modeling and Computation* 4, pp. 75–97.

Biere, Armin (2014). «Challenges in Bit-Precise Reasoning». *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*. FMCAD '14. Lausanne, Switzerland: FMCAD Inc, 2:3–2:3. ISBN: 978-0-9835678-4-4.

Biere, Armin, Florian Lonsing, and Martina Seidl (2011). «Blocked Clause Elimination for QBF». *International Conference on Automated Deduction*. Vol. 6803. Lecture Notes in Computer Science. Springer. Berlin, Heidelberg, pp. 101–115. DOI: 10.1007/978-3-642-22438-6_10.

Blanchette, Jasmin Christian and Tobias Nipkow (2010). «Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder». *International Conference on Interactive Theorem Proving*. Springer, pp. 131–146. DOI: 10.1007/978-3-642-14052-5_11.

Bogaerts, Bart, Tomi Janhunen, and Shahab Tasharrofi (2016). *Solving QBF Instances with Nested SAT Solvers*.

Bonichon, Richard, David Déharbe, Pablo Dobal, and Cláudia Tavares (2015). «SMTpp : preprocessors and analyzers for SMT-LIB».

Boolos, George (1987). «A Curious Inference». *Journal of Philosophical Logic* 16.1, pp. 1–12.

Bouton, Thomas, Diego Caminha B. De Oliveira, David Déharbe, and Pascal Fontaine (2009). «veriT: An Open, Trustable and Efficient SMT-Solver». *Proceedings of the 22Nd International Conference on Automated Deduction*. CADE-22. Montreal, P.Q., Canada: Springer, pp. 151–156. DOI: 10.1007/978-3-642-02959-2_12.

Brown, Chad E. (2013). «Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems». *Journal of Automated Reasoning* 51.1, pp. 57–77. DOI: 10.1007/s10817-013-9283-8.

Bryant, Randal E. and Miroslav N. Velev (2000). «Boolean Satisfiability with Transitivity Constraints». *Computer Aided Verification: 12th International Conference. Proceedings*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer, pp. 85–98. DOI: 10.1007/10722167_10.

Büning, Hans Kleine and Uwe Bubeck (2009). «Theory of Quantified Boolean Formulas». *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn J.H. Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. Amsterdam, The Netherlands: IOS Press. Chap. 23, pp. 735–760. DOI: 10.3233/978-1-58603-929-5-825.

Büning, Hans Kleine, Karpinski Karpinski, and Flögel Flogel (1995). «Resolution for Quantified Boolean Formulas». *Information and Computation* 117.1, pp. 12–18. DOI: 10.1006/inco.1995.1025.

Church, Alonzo (1940). «A formulation of the simple theory of types». *The Journal of Symbolic Logic* 5.2, pp. 56–68. DOI: 10.2307/2266170.

Devriendt, Jo, Bart Bogaerts, and Maurice Bruynooghe (2014). «BreakIDGlucose: On the importance of row symmetry in SAT». *Proceedings 4th International Workshop on the Cross-Fertilization Between CSP and SAT*, pp. 1–17.

Eén, Niklas and Armin Biere (2005). «Effective Preprocessing in SAT Through Variable and Clause Elimination». *Theory and Applications of Satisfiability Testing: 8th International Conference. Proceedings*. Ed. by Fahiem Bacchus and Toby Walsh. Berlin, Heidelberg: Springer, pp. 61–75. DOI: 10.1007/11499107_5.

Egly, Uwe, Martina Seidl, Hans Tompits, Stefan Woltran, and Michael Zolda (2004). «Comparing Different Prenexing Strategies for Quantified Boolean Formulas». *Theory and Applications of Satisfiability Testing – SAT 2004*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer, pp. 214–228. DOI: 10.1007/978-3-540-24605-3_17.

Enderton, Herbert Bruce (2015). «Second-order and Higher-order Logic». *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2015. Metaphysics Research Lab, Stanford University.

Finkbeiner, Bernd and Leander Tentrup (2014). «Theory and Applications of Satisfiability Testing - SAT 2014. Proceedings». *SAT*. Ed. by Carsten Sinz and Uwe Egly. Vol. 8561. Lecture Notes in Computer Science. Springer, pp. 243–251. DOI: 10.1007/978-3-319-09284-3_19.

Fitting, Melvin (1996). *First-Order Logic and Automated Theorem Proving*. 2nd ed. Texts in Computer Science. New York: Springer. DOI: 10.1007/978-1-4612-2360-3.

Fröhlich, Andreas, Gergely Kovásznai, and Armin Biere (2012). «A DPLL Algorithm for Solving DQBF». *Proceedings of the International Workshop on Pragmatics of SAT*. Trento, Italy.

Fröhlich, Andreas, Gergely Kovásznai, Armin Biere, and Helmut Veith (2014). «iDQ: Instantiation-Based DQBF Solving.» *Proceedings of the International Workshop on Pragmatics of SAT*.

Gitina, Karina, Sven Reimer, Matthias Sauer, Ralf Wimmer, Christoph Scholl, and Bernd Becker (2013). «Equivalence checking of partial designs using dependency quantified Boolean formulae». *International Conference on Computer Design*. IEEE, pp. 396–403. DOI: 10.1109/ICCD.2013.6657071.

Gitina, Karina, Ralf Wimmer, Sven Reimer, Matthias Sauer, Christoph Scholl, and Bernd Becker (2015). «Solving DQBF Through Quantifier Elimination». *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. DATE '15. Grenoble, France: EDA Consortium, pp. 1617–1622. ISBN: 978-3-9815370-4-8.

Giunchiglia, Enrico, Paolo Marin, and Massimo Narizzano (2010). «sQueezeBF: An Effective Preprocessor for QBFs Based on Equivalence Reasoning». *Theory and Applications of Satisfiability Testing – SAT 2010*. Ed. by Ofer Strichman and Stefan Szeider. Berlin, Heidelberg: Springer, pp. 85–98. DOI: 10.1007/978-3-642-14186-7_9.

Gödel, Kurt (1930). «Die Vollständigkeit der Axiome des logischen Funktionenkalküls». *Monatshefte für Mathematik und Physik* 37.1, pp. 349–360. DOI: 10.1007/BF01696781.

Gödel, Kurt (1931). «Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I». *Monatshefte für Mathematik und Physik*

38.1, pp. 173–198. DOI: 10.1007/BF01700692.
Henkin, Leon (1950). «Completeness in the Theory of Types». *The Journal of Symbolic Logic* 15.2, pp. 81–91. DOI: 10.2307/2266967.
Heule, Marijn J.H., Martina Seidl, and Armin Biere (2014). «A Unified Proof System for QBF Preprocessing». *International Joint Conference on Automated Reasoning*. Vol. 8562. Lecture Notes in Computer Science. Springer, pp. 91–106. DOI: 10.1007/978-3-319-08587-6_7.
Hoder, Krystof, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov (2012). «Preprocessing techniques for first-order clausification». *Proceedings of the 12th Conference on Formal Methods in Computer-Aided Design*. Cambridge, UK: IEEE, pp. 44–51. ISBN: 978-0-9835678-2-0.
Huet, Gerard Pierre (1972). «Constrained Resolution: A Complete Method for Higher-order Logic.» PhD thesis. Cleveland, OH, USA.
Huet, Gerard Pierre (1975). «A unification algorithm for typed λ-calculus». *Theoretical Computer Science* 1.1, pp. 27–57. DOI: 10.1016/0304-3975(75)90011-0.
Janota, Mikoláš, Charles Jordan, Will Klieber, Florian Lonsing, Martina Seidl, and Allen Van Gelder (2016). «The QBFGallery 2014: The QBF competition at the FLoC olympic games». *Journal on Satisfiability, Boolean Modeling and Computation* 9, pp. 187–206.
Janota, Mikoláš, William Klieber, Joao Marques-Silva, and Edmund Clarke (2012). «Solving QBF with counterexample guided refinement». *Theory and Applications of Satisfiability Testing – SAT 2012*. Springer, pp. 114–128. DOI: 10.1016/j.artint.2016.01.004.
Janota, Mikoláš and Joao Marques-Silva (2015). «Expansion-based QBF solving versus Q-resolution». *Theoretical Computer Science* 577, pp. 25–42. DOI: 10.1016/j.tcs.2015.01.048.
Järvisalo, Matti, Armin Biere, and Marijn J.H. Heule (2010). «Blocked Clause Elimination». *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Javier Esparza and Rupak Majumdar. Berlin, Heidelberg: Springer, pp. 129–144. DOI: 10.1007/978-3-642-12002-2_10.
Järvisalo, Matti, Marijn J.H. Heule, and Armin Biere (2012). «Inprocessing Rules». *Automated Reasoning*, pp. 355–370. DOI: 10.1007/978-3-642-31365-3_28.
Kiesl, Benjamin, Martin Suda, Martina Seidl, Hans Tompits, and Armin Biere (2017). «Blocked Clauses in First-Order Logic». *ArXiv e-prints*. arXiv: 1702.00847 [cs.LO].
Knight, Kevin (1989). «Unification: A Multidisciplinary Survey». *ACM Computing Surveys* 21.1, pp. 93–124. DOI: 10.1145/62029.62030.
Kovásznai, Gergely (2015). «A Survey on DQBF: Formulas, Applications, Solving Approaches». *QUANTIFY 2015*, p. 8.

Kullmann, Oliver (1999). «On a generalization of extended resolution». *Discrete Applied Mathematics* 96, pp. 149–176. DOI: 10.1016/S0166-218X(99)00037-2.

Lonsing, Florian, Fahiem Bacchus, Armin Biere, Uwe Egly, and Martina Seidl (2015). «Enhancing search-based QBF solving by dynamic blocked clause elimination». *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, pp. 418–433. DOI: 10.1007/978-3-662-48899-7_29.

Lonsing, Florian and Armin Biere (2010). «DepQBF: A dependency-aware QBF solver». *Journal on Satisfiability, Boolean Modeling and Computation* 7, pp. 71–76.

Miller, Dale (1987). «A compact representation of proofs». *Studia Logica* 46.4, pp. 347–370. DOI: 10.1007/BF00370646.

Moschovakis, Joan (2015). «Intuitionistic Logic». *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2015. Metaphysics Research Lab, Stanford University.

Moura, Leonardo de and Nikolaj Bjørner (2008). «Z3: An Efficient SMT Solver». *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference. Proceedings*. Ed. by C.R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

Niemetz, Aina, Mathias Preiner, and Armin Biere (2014). «Boolector 2.0 system description». *Journal on Satisfiability, Boolean Modeling and Computation* 9, pp. 53–58.

Nipkow, Tobias (1993). «Functional Unification of Higher-Order Patterns». *Proceedings of Eighth Annual IEEE Symposium on Logic in Computer Science*. IEEE, pp. 64–74. DOI: 10.1109/LICS.1993.287599.

Nipkow, Tobias, Lawrence C. Paulson, and Markus Wenzel (2002). *Isabelle/-HOL – A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. ISBN: 3-540-43376-7.

Parikh, Rohit Jivanlal (1973). «Some results on the length of proofs». *Transactions of the American Mathematical Society* 177, pp. 29–36. DOI: 10.1090/S0002-9947-1973-0432416-X.

Paulson, Lawrence C. and Jasmin Christian Blanchette (2010). «Three Years of Experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers». *IWIL@LPAR*. Vol. 2. EPiC Series.

Peterson, Gary L. and John H. Reif (1979). «Multiple-person Alternation». *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, pp. 348–363. DOI: 10.1109/SFCS.1979.25.

Pigorsch, Florian and Christoph Scholl (2010). «An AIG-Based QBF-solver Using SAT for Preprocessing». *Proceedings of the 47th Design Automation*

Conference. DAC '10. Anaheim, California: ACM, pp. 170–175. DOI: 10.1145/1837274.1837318.

Pulina, Luca (2016). «The Ninth QBF Solvers Evaluation – Preliminary Report». *Proceedings of the 4th International Workshop on Quantified Boolean Formulas (QBF 2016)*. Ed. by Florian Lonsing and Martina Seidl. Bordeaux, France.

Robinson, John Alan (1965). «A Machine-Oriented Logic Based on the Resolution Principle». *Journal of the ACM* 12.1, pp. 23–41. DOI: 10.1145/321250.321253.

Seidl, Martina, Florian Lonsing, and Armin Biere (2012). «qbf2epr: A Tool for Generating EPR Formulas from QBF». *Third Workshop on Practical Aspects of Automated Reasoning*, p. 139.

Shostak, Robert E. (1978). «An Algorithm for Reasoning About Equality». *Communications of the ACM* 21.7, pp. 583–585. DOI: 10.1145/359545.359570.

Steen, Alexander, Max Wisniewski, and Christoph Benzmüller (2016). «Agent-Based HOL Reasoning». *Mathematical Software – ICMS 2016*. Ed. by Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew Sommese. Vol. 9725. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 75–81. DOI: 10.1007/978-3-319-42432-3_10.

Steen, Alexander, Max Wisniewski, and Christoph Benzmüller (2017). «Going Polymorphic - TH1 Reasoning for Leo-III». *IWIL@LPAR 2017 Workshop and LPAR-21 Short Presentations*. Ed. by Thomas Eiter, David Sands, Geoff Sutcliffe, and Andrei Voronkov. Vol. 1. Kalpa Publications in Computing. Maun, Botswana: EasyChair.

Sutcliffe, Geoff (2009). «The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0». *Journal of Automated Reasoning* 43.4, pp. 337–362. DOI: 10.1007/s10817-009-9143-8.

Sutcliffe, Geoff and Christoph Benzmüller (2010). «Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure». *Journal of Formalized Reasoning* 3.1, pp. 1–27. ISSN: 1972-5787.

Sutcliffe, Geoff, Jürgen Zimmer, and Stephan Schulz (2003). «Communication Formalisms for Automated Theorem Proving Tools». *Proceedings of the Workshop on Agents and Automated Reasoning, 18th International Joint Conference on Artificial Intelligence*, pp. 52–57.

Tseitin, Grigori Samuilovich (1983). «On the Complexity of Derivation in Propositional Calculus». *Automation of reasoning*. Berlin Heidelberg: Springer, pp. 466–483. DOI: 10.1007/978-3-642-81955-1_28.

Wimmer, Ralf, Karina Gitina, Jennifer Nist, Christoph Scholl, and Bernd Becker (2015). «Preprocessing for DQBF». *Theory and Applications of Satisfiability Testing – SAT 2015*. Ed. by Marijn J.H. Heule and Sean Weaver.

Vol. 9340. Lecture Notes in Computer Science. Springer, pp. 173–190. DOI: 10.1007/978-3-319-24318-4_13.

Wimmer, Ralf, Sven Reimer, Paolo Marin, and Bernd Becker (2017). «HQSpre – An Effective Preprocessor for QBF and DQBF». *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference. Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Berlin, Heidelberg: Springer, pp. 373–390. DOI: 10.1007/978-3-662-54577-5_21.

Wisniewski, Max, Alexander Steen, and Christoph Benzmüller (2014). «The Leo-III Project». *Joint Automated Reasoning Workshop and Deduktionstreffen*. Ed. by Alexander Bolotov and Manfred Kerber, p. 38.

Wisniewski, Max, Alexander Steen, and Christoph Benzmüller (2015). «LeoP-ARD – A Generic Platform for the Implementation of Higher-Order Reasoners». *Intelligent Computer Mathematics – CICM 2015*. Ed. by Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge. Vol. 9150. Lecture Notes in Computer Science. Springer, pp. 325–330. DOI: 10.1007/978-3-319-20615-8_22.

Wisniewski, Max, Alexander Steen, Kim Kern, and Christoph Benzmüller (2016). «Effective Normalization Techniques for HOL». *Automated Reasoning – 8th International Joint Conference*. Ed. by Nicola Olivetti and Ashish Tiwari. Vol. 9706. Lecture Notes in Computer Science. Springer, pp. 362–370. DOI: 10.1007/978-3-319-40229-1_25.

Index