Bachelor thesis at the Institute of Computer Science

Research group Leo-III

Improved Computation of CNF in Higher-Order Logics

Kim Kern
Matriculation number: 4467609
kim.kern@fu-berlin.de

Supervisor: PD Dr. Christoph Benzmüller
Second examiner: Prof. Dr. Elfriede Fehr

Berlin, 19.05.2015

Abstract

The computation of clause normal forms (CNF) is an essential preprocessing step in automated reasoning. With effective methods, not only the computation itself, but also its results, can be improved in terms of their complexity, and therefore allow for shorter proofs. Although research in first-order contexts has already shown the potential of such techniques, little has been published about this in higher-order logics thus far.

This thesis provides a basic implementation of the conjunctive normal form computation in a higher-order context. Therefore, a common first-order approach is examined for possible portability; some combinable improving techniques are presented as well.

Acknowledgment

I'd like to thank my supervisor, Dr. Christoph Benzmueller, and the whole team around Leo-III, namely Max Wisniewski and Alexander Steen, for their great support and encouragement.

Declaration of authorship

I hereby certify, that I have written this thesis entirely on my own and that I have not used any other materials than the ones referred to. This thesis or parts of it have not been submitted for a degree at this or any other university.

19.05.2015

Kim Kern

Contents

1 Introduction

1.1 Historical Context

The research of logics goes way back to Ancient Greece with Aristoteles (384-322 BC) and his Syllogisms [AvK08]. The concept of formalizing arguments has remained present during history and has been utilized by other important philosophers such as Leibniz (1646-1716). Leibniz was following the idea of the interaction of a formal language (Characteristica universalis) and a reasoning system (Calculus ratiocinator) to describe philosophical problems and then calculate their correctness [Lei65].

This was followed by a milestone for modern logics in 1879, when Gottlob Frege (1848-1925) published his work *Begriffsschrift* [Fre64]. In that Gottlob introduced the first formalization of a second order (and therefore higher-order) predicate logic. Later *Russell's Paradox* (1901) showed that untyped quantification over predicates as in the *Begriffsschrift* results in inconsistencies. But Frege also established the theory of Logicism which fascinated a lot of mathematicians and lead to the *Principia Mathematica* (1910-1913) [WR12] by Alfred Whitehead and Betrand Russell. They pursued the aim to derive all mathematical truths from a logical system, which was later proven impossible by Kurt Gödel's (1906-1978) famous *incompleteness theorems* (1931) [Göd31]. It was shown that any sufficiently complex system is either incomplete or unsound, by which the mathematical community around David Hilbert (1862-1943) was shaken.

In 1940 Alonzo Church presented his *Simple Theory of Types* [Chu40], which is much simpler and more general than Russell's type theory. Although it is often referred to as classical higher-order logic [oP15], it was still struggling with the incompleteness issue. Leon Henkin's (1921-2006) research in 1950 allowed for the first complete calculus in higher-order logic introducing the more general *Henkin semantics* [Hen50].

1.2 Higher-Order Logics

As previously mentioned, Russell created a paradox, with that he constructed a contradiction by defining a set which both does and does not contain itself. So, he could show that the untyped lambda calculus is inconsistent because of its expressiveness.

That's why higher-order logics (HOL) are based on Church's Simple Theory of Types [Chu40]. Here not only is the paradox avoided, it's also guaranteed that every term has a $\beta\eta$-normal form — a unique representation for a formula. The expressiveness is increased through quantification over functions, sets and relations of any types and unnamed functions, sets and relations

[BM14]. Therewith, for example, one can express the following condition which is not expressible in a first-order context: "It exists a function with two zeros.", however with the gained expressiveness comes the incompleteness of all calculi for full Simple Type Theory in standard semantics [Göd31]. Conversely, the usage of the more general Henkin semantics [Hen50] allowed for the definition of already several complete calculi.

That's why this work is developed for higher-order logics with Henkin semantics. Additionally, the Axiom of Choice [And72, BM14] is assumed. This choice corresponds to the theory framework of the THF0 TPTP language [BRS08, Sut09].

Higher-order logics are used in philosophy (e.g. formal proof for Gödel's ontological proof [Göd95]), formal software verification or automated proof validation for paper reviews. Also, other logics such as modal logics (e.g. quantified conditional logics [Ben13]) — for which no automated provers exist so far — can be simulated in higher-order logics.

Still research in higher-order logics is not as advanced as for first-order logics, however interesting work has already been published and there is bound to be more to follow.

1.3 Leo-III

Leo-III[1] is an automated proving system for higher-order logics [WSB14]. Earlier versions of the Leo HOL prover already had implemented cooperation to external systems, as for example E [Sch04] for first-order proofs in Leo-II [BTPF08]. Also Leo-II was embedded in the proof assistant system ISABELLE [NPW02]. Leo-III will now increase the cooperation between provers by implementing massive parallelism (And/Or-Parallelism, Multisearch) with a multi-agent blackboard architecture [Wis14]. With it, other existing proof systems and the TPTP infrastructure [Sut09] are reused. It also features a much more expressive data structure, which supports type polymorphism [Ste14].

1.4 Motivation and Aims of Thesis

A conjunctive normal form or clause normal form (CNF) is a conjunction of clauses where clauses consist of disjunctions of literals, which can also be represented as a set of sets of literals. The CNF normalization of a formula is a preprocessing step in automated reasoning to improve the efficiency of proof finding procedures.

[1] Leo-III website: http://page.mi.fu-berlin.de/lex/leo3/

As Peter Andrews points out, the repeated use of the distributive law in the computation of clause normal forms can cause "wild proliferation of literals" [And81]. Hence, optimization techniques must be found to reduce the application of the distributive law and therewith the resulting number of clauses. Therefore, some existing methods in first-order context are checked for possible portability to higher-order logics and new practises are presented to motivate the various possibilities for improvements.

Although the research on higher-order provers is getting increasingly more sophisticated, still no publications exist on CNF computation in a higher-order context.

The algorithm provided in this thesis has a dominantly modular structure, thus making use of the agent-based framework in Leo-III. Thereby the potential parallelization can be applied and single modules can be reused. Furthermore, the implementation can be augmented by the usage of flags and also eventually machine learning concepts to improve the effect by taking different measures such as problem size, domain or structure into account.

2 Preliminaries

Simple Type Theory (STT), for which *higher-order logics* are used as a synonym from now on, is based on Church's simply typed λ-calculus [Chu40] and usually uses the *basic set of types* $T = \{o, \iota\}$, where type o denotes Booleans and ι a domain of individuals.

Types The *simple set of types* $\mathcal{T}$ is constructed with the function type constructor $\to$, such that it is the smallest set containing the basic types in T and all type expressions $(\gamma \to \tau)$, where γ and τ are type expressions themselves [BM14]. For better readability, the type $\alpha \to (\beta \to \gamma)$ is denoted with the left-associative notation $\gamma\beta\alpha$, following common conventions. In this thesis, the types of the formulae are omitted if they can be inferred or aren't required for the understanding.

Terms A typed term is constructed from the following sets and operations:

X_α	Variables
c_α	Constants and Parameters
$(F_{\beta\alpha}\, B_\alpha)_\beta$	Application
$(\lambda Y_\alpha . A_\beta)_{\beta\alpha}$	λ-abstraction

Application associates to the left and abstraction to the right. The dot in $(\lambda Y_\alpha . A_\beta)_{\beta\alpha}$ means that the longest possible following part of the formula is in brackets without changing existing brackets. So $\lambda X.(F\,(\lambda Y.(A)))$ can be denoted as $\lambda X.F\,(\lambda Y.A)$. Also note that the spine notation is used for the application of terms, hence $(F\,X\,Y\,Z)$ represents $(((F\,X)\,Y)\,Z)$ [CP03].

The names of bound variables in the λ-statements are interchangeable with regard to α-conversion. That's why it's common practice to use the nameless binding mechanism called *de Bruijn index* for the implementation of bound variables [DB72]. This means there is no longer the requirement to rename according to α-conversion.

Logical connectives The *primitive logical connectives* $\neg_{oo}$ (not), $\vee_{oo}$ (or) and $\Pi_{o(o\alpha)}$ (quantification over type α) are used to construct formulae.

The following logical constants and connectives are defined as usual [Ben15].

$\top_o$	(truth)	$\Leftrightarrow_{ooo}$	(equivalence)
$\bot_o$	(falsehood)	$\forall$	(universal quantification)
$\wedge_{oo}$	(and)	$\exists$	(existential quantification)
$\supset_{ooo}$	(implication)	$=_{o\alpha\alpha}$	(equality at type α)

Table 1: Defined logical connectives

For some logical standard functions the common infix notation is used, e.g. $\phi \vee \psi$ standing for the application $(\vee_{ooo}\, \phi_o)\, \psi_o$. In a higher-order context Π in combination with the λ-operator can be utilized to construct the universal and existential quantifiers. Because the λ-operator already offers a binding mechanism, it can be reused for the quantification. So $\forall X.\phi$ is defined as $\Pi^{\alpha}_{o(o\alpha)}\lambda X_{\alpha}.\phi$ and respectively $\exists X.\phi$ as $\neg\Pi^{\alpha}_{o(o\alpha)}\lambda X_{\alpha}.\neg\phi$.

Free variables All variables that occur in a formula ϕ shall be called $vars(\phi)$. The set $bound(\phi)$ contains all bound variables in the formula ϕ. A variable in ϕ is called bound if it is within the scope of a λ that binds it. The free variables of a formula ϕ are defined as $free(\phi) = vars(\phi) \setminus bound(\phi)$.

Replacing $\omega[\phi/\psi]$ denotes the formula where all occurrences of ϕ in ω are replaced by ψ. The formula obtained, when the subformula at position π of the formula ψ is replaced by ϕ, is labelled $\psi[\pi/\phi]$.

Equality Equality of terms is defined by the following conversions that build a unique normal form:

α-conversion	Rename bound variables
β-reduction	$((\lambda Y_{\beta}.A_{\alpha})B_{\beta}) \rightarrow A[Y/B]$
η-reduction	$(\lambda Y_{\alpha}.(F_{\beta\alpha}Y)) \rightarrow F$, $(Y_{\alpha} \notin free(F))$

Polarity The *polarity* of a formula ϕ at position π is given by $pol(\phi, \pi)$, where $\phi \equiv \psi$ denotes that ϕ and ψ are equal:

$$pol(\phi,\epsilon) = \begin{cases} -1 & if\ \phi|_0 \equiv \neg\ or\ \phi|_0 \equiv\supset \\ 0 & if\ \phi|_0 \equiv\Leftrightarrow \\ 1 & else \end{cases}$$

$$pol(\phi,\pi.i) = \begin{cases} -pol(\phi,\pi) & if\ \phi|_{\pi 0} \equiv \neg\ or\ \phi|_{\pi 0} \equiv\supset \\ 0 & if\ \phi|_{\pi 0} \equiv\Leftrightarrow \\ pol(\phi,\pi) & else \end{cases}$$

Leibniz equality Leibniz equality $A_{\tau} \doteq^{\tau} B_{\tau}$ for two terms over type τ is defined by the formula $\forall P_{o\tau}.P\,A \supset P\,B$ [Ben15]. So two terms are equal if all predicates over that type result in the same value. It is sufficient to prove only one direction for equality.

Boolean extensionality Boolean extensionality $P \Leftrightarrow Q \supset P \doteq^{o} Q$ states that two Booleans are equal if they are equivalent. [BBK04]

Functional extensionality Functional extensionality $(\forall X.F\,X \doteq G\,X) \supset F \doteq^{\gamma\tau} G$ applies if and only if for any argument the corresponding function value is the same.

Positions A *position* π is a sequence of natural numbers. All possible positions of a formula ϕ are in the set $pos(\phi)$. $\psi[\phi]_\pi$ is short for $\psi|_\pi = \phi$ and $\psi[\pi/\phi]$ stands for the formula ψ where the subformula at position π was replaced by ϕ.

The formula $\phi|_\pi$ at position π of the formula ϕ is defined as follows:

π	$\phi\|_\pi$
ϵ	ϕ
$\tau 0$	F, if $\phi\|_\tau = (F\,X_n)$ and $X_n = x_1 \ldots x_n$
	A, if $\phi\|_\tau = \lambda Y.A$
	ψ, if $\phi\|_\tau = \forall X.\psi$ or $\phi_\tau = \exists X.\psi$
τi	x_i, if $\phi\|_\tau = (F\,X_n)$ and $X_n = x_1 \ldots x_n$

Table 2: Positions π

This can be illustrated with a short example: Let $\phi = \exists X.\neg Z \vee (X \wedge Y)$ and $\psi = (F\,X\,Y\,Z)$ be two formulae. Amongst others the subformulae $\phi|_{021} = X \wedge Y$, $\phi|_{00} = \neg$, $\psi_0 = F$ and $\psi_1 = X$ can now be addressed.

Depth The maximum length of a formula's position is called its *depth*. $depth(\phi) = max\{|\pi| \mid \pi \in pos(\pi)\}$

3 Standard CNF Algorithm

This section describes to what extent the basic algorithm given for first-order problems in [NW01] can be applied to higher-order logics.

First, unnecessary redundancies are removed in the *simplification* algorithm. The *negation normal form* guarantees that negations can only be found in front of literals. As a preprocessing step for the following *Skolemization, miniscoping* and *variable renaming* ensure, that quantifiers only appear at a innermost position and bound variables have distinct representations. Existential quantifiers are then removed using *Skolemization.* Finally, all universal quantifiers are removed and a *clause normal form* is constructed, on which a *clause simplification* is performed.

3.1 Simplification

The simplification algorithm removes unnecessary redundancies and obvious tautologies. The procedure provides simple replacement rules that can be exhaustively applied to higher-order formulae in any order. The rules are equivalence preserving and always terminate since the left hand side is smaller.

The algorithm can be applied in a higher-order context without any problems. The easiest way to implement this is by traversing the formula top-down and check for the rule conditions on every level.

$\phi \wedge \phi$	$\rightarrow \phi$	$\phi \vee \phi$	$\rightarrow \phi$
$\phi \wedge \neg\phi$	$\rightarrow \bot$	$\phi \vee \neg\phi$	$\rightarrow \top$
$\phi \Leftrightarrow \phi$	$\rightarrow \top$	$\phi \supset \phi$	$\rightarrow \top$
$\neg\top$	$\rightarrow \bot$	$\neg \bot$	$\rightarrow \top$
$\phi \wedge \top$	$\rightarrow \phi$	$\phi \vee \top$	$\rightarrow \top$
$\phi \wedge \bot$	$\rightarrow \bot$	$\phi \vee \bot$	$\rightarrow \phi$
$\phi \supset \top$	$\rightarrow \top$	$\top \supset \phi$	$\rightarrow \phi$
$\phi \supset \bot$	$\rightarrow \neg\phi$	$\bot \supset \phi$	$\rightarrow \top$
$\phi \Leftrightarrow \top$	$\rightarrow \phi$	$\phi \Leftrightarrow \bot$	$\rightarrow \neg\phi$

$\forall X.\phi \rightarrow \phi,\ if\ x \notin free(\phi)$
$\exists X.\phi \rightarrow \phi,\ if\ x \notin free(\phi)$

Table 3: Simplification

3.2 Negation Normal Form

For the negation normal form equivalences $\Leftrightarrow$, implications $\supset$ and double negations $\neg\neg$ need to be removed while making sure that negations $\neg$ are only found in front of atoms. The given rules are again terminating and equivalence preserving. The removal of equivalences is *polarity dependent* so that the construction of tautologies can be avoided.

$\psi[\phi_1 \Leftrightarrow \phi_2]_\pi$	$\rightarrow$	$\psi[\pi/(\phi_1 \supset \phi_2) \wedge (\phi_2 \supset \phi_1)]$	$if\, pol(\psi, \pi) = 1$		
$\psi[\phi_1 \Leftrightarrow \phi_2]_\pi$	$\rightarrow$	$\psi[\pi/(\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge \neg\phi_2)]$	$if\, pol(\psi, \pi) = -1$		
$\neg(\phi \wedge \psi)$	$\rightarrow$	$\neg\phi \vee \neg\psi$	$\neg(\phi \vee \psi)$	$\rightarrow$	$\neg\phi \wedge \neg\psi$
$\neg\forall X.\phi$	$\rightarrow$	$\exists X.\neg\phi$	$\neg(\exists X.\phi)$	$\rightarrow$	$\forall X.\neg\phi$
$\phi \supset \psi$	$\rightarrow$	$\neg\phi \vee \psi$	$\neg\neg\phi$	$\rightarrow$	ϕ

Table 4: Negation normal form

The initial polarity of the formula ϕ must be considered because it has an impact on the whole formula. If the polarity of ϕ is $pol(\phi) = -1$, the algorithm should be run with $\neg\phi$.

3.3 Miniscoping

Before existential quantifiers are removed (see Skolemization, section 3.5), all quantifiers shall be moved the furthest inwards. This minimizes the arity of the Skolem functions, which will be explained in the coming passages.

$\exists X.\phi \wedge \psi$	$\rightarrow$	$(\exists X.\phi) \wedge \psi$	if $X \notin free(\psi)$
$\exists X.\phi \vee \psi$	$\rightarrow$	$(\exists X.\phi) \vee \psi$	if $X \notin free(\psi)$
$\forall X.\phi \wedge \psi$	$\rightarrow$	$(\forall X.\phi) \wedge \psi$	if $X \notin free(\psi)$
$\forall X.\phi \vee \psi$	$\rightarrow$	$(\forall X.\phi) \vee \psi$	if $X \notin free(\psi)$
$\exists X.\phi \vee \psi$	$\rightarrow$	$(\exists X.\phi) \vee (\exists X.\psi)$	if $X \in free(\phi)$ and $X \in free(\psi)$
$\forall X.\phi \wedge \psi$	$\rightarrow$	$(\forall X.\phi) \wedge (\forall X.\psi)$	if $X \in free(\phi)$ and $X \in free(\psi)$

Table 5: Miniscoping

It is not obvious why quantifiers wouldn't be moved inwards in function applications. The following illustrates why that wouldn't be sound in general. The formula $\exists X.f(X)$ is considered where X is of type ι and f a function of type $\alpha\iota$. If the quantifier is moved inwards, the argument of the function would be of type o, and thus not match the signature any more.

If a naive application of those rules is performed, formulae might be encountered, where at first glance none of the above rules seem to apply. But when a small equivalence preserving transformation is applied, it may help to move the quantifiers further inwards.

Consider the formula $\phi = \forall X.p(X) \wedge (q(X) \wedge p(Y))$. Since for both subformulae $\phi_1 = \phi|_{01}$ and $\phi_2 = \phi|_{02}$ holds $X \in free(\phi_i)$, $i = 1, 2$ none of the rules stated before can be applied. But if the formula is transformed according to the associative law to $\forall X.(p(X) \wedge p(X)) \wedge p(Y)$ it can make use of the rules as follows:

$$\forall X.(p(X) \wedge q(X)) \wedge p(Y)$$
$$\Downarrow$$
$$(\forall X.p(X) \wedge q(X)) \wedge p(Y)$$
$$\Downarrow$$
$$((\forall X.p(X)) \wedge (\forall X.q(X))) \wedge p(Y)$$

Also the order of the quantifiers may make a difference. This can be demonstrated with the formula $\phi = \forall X.\forall Y.f(X, Y) \wedge f(Y, Y)$. Since again for both subformulae $\phi_1 = \phi|_{001}$ and $\phi_2 = \phi|_{002}$ $X \in free(\phi_i)$, $i = 1, 2$ applies, the quantifier $\forall Y$ can't be moved further inwards. If one switches the two quantifiers however, it can be proceeded as follows:

$$\forall Y.\forall X.f(X, Y) \wedge f(Y, Y) \rightarrow \forall Y.(\forall X.f(X, Y)) \wedge f(Y, Y)$$

3.4 Variable Renaming

The Miniscoping algorithm in section 3.3 duplicates quantifiers and thus also their bound variables. Since the universal quantifiers will be moved outwards for the Clause Normal Form in section 3.6, we would run into a name conflict. Yet, variable bindings in Leo-III are represented using de Bruijn indices [DB72]. Because of their nameless representation, those name conflicts can be avoided in a very elegant manner and therefore make this step unnecessary. However, it is crucial to keep track of the indices and possibly raise or decrease them, when changing the positions of the binding quantifiers.

It should be kept in mind though that, in general, existential and universal quantifiers are not arbitrarily interchangeable $\forall X.\exists Y.\phi \not\equiv \exists Y.\forall X.\phi$.

3.5 Skolemization

Skolemization is used to obtain a formula without existential quantifiers. For that, a new Skolem function is introduced for each existentially quantified variable which depends on the free variables in the scope of the corresponding quantifier. If there are no free variables in the scope of the existential quantifier, the thus created Skolem function with arity 0 is referred to as a Skolem constant.

	$\phi[\exists X.\psi]_\pi \rightarrow \phi[\pi/\phi[X/f(Y_1,...,Y_n)]]$
where	$free(\phi) = \{Y_1,...,Y_n\}$
	$f\, new\, skolem\, function\, of\, arity\, n$

Table 6: Skolemization

Skolemization doesn't preserve equivalence but satisfiability.

Without the prior application of miniscoping, the proof complexity may increase. The following example illustrates, how miniscoping can reduce the arity of the Skolem functions.

$$\exists X.X \vee Y \overset{Skolemization}{\Longrightarrow} f(Y) \vee Y$$

$$\exists X.X \vee Y \overset{miniscoping}{\Longrightarrow} (\exists X.X) \vee Y \overset{Skolemization}{\Longrightarrow} a \vee Y$$

In a higher-order context Skolemization can be applied as proposed in the first-order solution. The type of the introduced Skolem function or constant must match the type of the quantified variable.

3.6 Clause Normal Form

In this section, all accessible universal quantifiers are moved outwards and disjunctions are moved inwards, utilising the following three rules:

$(\forall X.\phi) \wedge \psi$	$\rightarrow\ \forall X.\phi \wedge \psi$	$if\ X \notin free(\psi)$
$(\forall X.\phi) \vee \psi$	$\rightarrow\ \forall X.\phi \vee \psi$	$if\ X \notin free(\psi)$
$\phi \vee (\psi_1 \wedge \psi_2)$	$\rightarrow\ (\phi \vee \psi_1) \wedge (\phi \vee \psi_2)$	

Table 7: Clause normal form

In higher-order logics a quantifier might still occur inside of an application and can therefore not be moved further outwards, e.g. $f_{\alpha o}(\forall X.\phi)$. The same applies to disjunctions, e.g. $f(a \wedge b) \vee c$. In section 4.2, a mechanism is described that replaces function arguments to reduce their depth to zero, and thus resolve the problem.

Furthermore in Leo-III, where de Bruijn indices [DB72] are used as binding mechanism, the condition $if\ X \notin free(\psi)$ is in this case always true. However, indices might have to be lifted when the quantifiers are moved outwards.

3.7 Clause Simplification

In first-order logics all universal quantifiers are removed, since they are all at the very outermost of the formula and all free variables are just considered to be universally quantified. In higher-order logics however, it is not guaranteed that no quantifiers can be found at positions $\pi \neq \epsilon$, because a quantifier might still be inside of an application like for instance in $f(\forall X.\phi)$. A method to move quantifiers the furthest outwards after all can be found in section 4.3.

A clause can be removed if it is a tautology, i.e. if a clause contains a literal and its corresponding negation.

4 Techniques for Higher-Order CNF

The computation of a clause normal form can result in a bloated set of clauses. Hence, in this section some techniques are presented, to reduce the number and size of the returned clauses.

4.1 Context Splitting

Context splitting [RV01] is used to decrease the problem size by splitting problems into subproblems. Therefore, it reduces the proof complexity and renders further parallelization possible. A disjunction $\omega = \phi \vee \psi$ (conjunction $\omega = \phi \wedge \psi$) is split into two separate proofs, if their sets of free variables are disjoint. The initial problem ω is true if one (both) of the subproblems is (are) true.

$\phi \wedge \psi \rightarrow t_1 \wedge t_2$	$if\ free(\phi) \cap free(\psi) = \emptyset$
$\phi \vee \psi \rightarrow t_1 \vee t_2$	$if\ free(\phi) \cap free(\psi) = \emptyset$

Table 8: Context splitting

Context splitting should be applied after miniscoping, since additional possibilities for further splittings may emerge as illustrated in the following example.

$\forall X.f(X) \wedge g(X)$	where $free(f(X)) \cap free(g(X)) = \{X\}$
$\Downarrow miniscoping$	
$\forall X.f(X) \wedge \forall X.g(X)$	where $free(\forall X.f(X)) \cap free(\forall X.g(X)) = \emptyset$

Because in the miniscoping algorithm, formulae must be checked for their free variables already, the two techniques can be combined for the implementation. The reduced computational expense comes at the price of a decreased encapsulation.

The proof search might also benefit from a splitting which is performed even before the normalization, since the normalization itself and the processing of the resulting problems can be parallelized. Further research is needed to determine appropriate indicators which suggest an effective application for a particular formula.

4.2 Bounded Argument Depth

In a higher-order context the arguments of applications can have an unrestricted depth, which leads to several difficulties. The problem shall be illustrated with the following example of a resolution proof:

$$p_{oo}(a) \wedge p(b) \supset p(a \wedge b) \tag{1}$$

$$\Downarrow \; normalization$$

$$\{\neg p(a),\ \neg p(b),\ p(a \wedge b)\}$$

$$\Downarrow \; resolution\, steps$$

$$\Box \vee p(a) \stackrel{?}{=} p(a \wedge b) \;\stackrel{dec}{\Longrightarrow}\; \Box \vee a \stackrel{?}{=} (a \wedge b)$$

$$\Box \vee p(b) \stackrel{?}{=} p(a \wedge b) \;\stackrel{dec}{\Longrightarrow}\; \Box \vee b \stackrel{?}{=} (a \wedge b)$$

After the decompositions, it can be seen that a basic unification algorithm [Hue75] would not be able to solve this problem and it would get stuck. But if a new normalization step is added, where the arguments' depth for all arguments of type o is reduced, a solution can easily be found. So every argument ϕ_o where $depth(\phi) > 1$ is replaced with a new definition $t = \phi$ and the definition is added to the set of clauses. With the unification constraint $\phi \stackrel{?}{=} \psi$, respectively as $\neg(\phi = \psi)$ and Boolean extensionality, the procedure results in the formulae $t \Leftrightarrow \phi$ and $\phi \Leftrightarrow \neg\psi$ [BBK04]:

$$f(\phi_o)\, and\, depth(\phi) > 1$$

$$\Downarrow t\, is\, a\, new\, symbol$$

$$f(\phi)[\phi/t]\, and\, addNewClause(t = \phi)$$

$$\phi_o \stackrel{?}{=} \psi_o \;\Longrightarrow\; addNewClause(\neg\phi \Leftrightarrow \psi).$$

Now the described procedure is applied to the given example. In a first normalization step $t \Leftrightarrow a \wedge b$ is added to the clauses set and then replaced $p(a \wedge b)$ by $p(t)$. After the decomposition this forms the unification constraints $a \stackrel{?}{=} t$ resp. $b \stackrel{?}{=} t$, which are both added to the clauses set as stated before. The set of clauses now contains the subset $S = \{(t \Leftrightarrow a \wedge b),\ (\neg t \Leftrightarrow a),\ (\neg t \Leftrightarrow b)\}$ It's obvious that S can be refuted, and thus the conjecture (1) has been proven.

The algorithm could be lifted to arbitrary argument types, such as oo, because functional extensionality can be applied. So in the function $p(f(a,b)_{oo})$

the argument can be replaced by $t_{oo} = f(a, b)$. Now functional extensionality can be applied so that $\forall X.t(X) \Leftrightarrow f(a, b)(X)$ is obtained, which can be appended to the set of clauses. It might be expedient to restrict this to types with a cardinality smaller than a still to be determined maximum.

Future works can evaluate whether building up an additional auxiliary structure as a *constraint store* to keep track of the newly created clauses can further improve the process by separating it from the rest of the proof and therefore enabling parallelism in problems with a reduced complexity.

4.3 Quantifier Extraction

In higher-order logics quantified formulae can be the argument of an application like in $f(\forall X.\phi)$ or $g((\exists X.\phi)\, Y)$. Therefore, they can't be moved outwards by standard normalization techniques. If the procedure described in the former section 4.2 is applied though, quantifiers can be moved to an outermost position and completely removed as explained in section 3.7. Contrary to the more general *Bounded Argument Depth*, the *Quantifier Extraction* can be performed prior to any presented normalization step.

In the following example, it is assumed that ϕ no longer contains any quantifiers at a position other than the outermost, because it has been normalized already. In the first step the application's argument is renamed according to the previous section. Since a quantified formula has been renamed, the new constant t is of type o. Thus, Boolean extensionality can be applied on the definition $t_o = \forall X.\phi$ so that $t \Leftrightarrow \forall X.\phi$ is obtained. Then the formula can be further normalized such that the quantifiers are moved to the outermost position after all. Since by this procedure all quantifiers inside of applications in ϕ could have been moved outwards as well, this is a general way to extract all quantifiers from within applications.

$$\psi[f(\forall X.\phi)]_\pi$$
$$\Downarrow rename\, arg.$$
$$\psi[f(t) \wedge (t_o = \forall X.\phi)]_\pi$$
$$\Downarrow Boolean\, ext.$$
$$\psi[f(t) \wedge (t \Leftrightarrow \forall X.\phi)]_\pi$$
$$\Downarrow normalize$$
$$quantifier\, can\, be\, moved\, outwards$$

If the condition $free(\forall X.\phi) = \emptyset$ applies for the argument in $\psi|_\pi = f(\forall X.\phi)$, the definition $t = \forall X.\phi$ can be added at the top level of ψ: $\psi[\pi/f(t)] \wedge$

$(t \Leftrightarrow \forall X.\phi)$ Therewith, possible applications of the distributivity law can be avoided and thus the number of resulting clauses reduced. This can't be applied if $free(\forall X.\phi) \neq \emptyset$ because in general $\exists X.f(a\,(\forall Y.X \wedge Y)) \not\equiv (\exists X.f(a\,t)) \wedge (t \Leftrightarrow X \wedge Y)$.

4.4 Formula Renaming

Formula renaming is an optimization concept proposed also in [NW01] to reduce the number of produced clauses before the normalization. When the conjunctive normal form is constructed, the number of clauses might explode due to the repetitive application of the distributive law. A formula $\omega = \phi \vee \psi$, where ϕ is normalized to n clauses and $\omega|_\pi = \psi$ to m clauses, the normalization of ω results in $n * m$ clauses. If ψ at position π is replaced by a definition r such that $\omega' = (\phi \vee r(X_1, ..., X_n)) \wedge def(\pi, \omega, r)$ where $free(\psi) = \{X_1, ..., X_n\}$ and the normalization of $def(\pi, \omega, r)$ yields in k clauses, ω' only generates $n * 1 + k$ clauses. How can be determined when the condition $n * m > n + k$ applies, hence a renaming reduces the number of clauses and what should the definition $def(\pi, \omega, r)$ look like?

If ψ is a formula and $\phi = \psi|_\pi$ the formula to be renamed, the formula $\psi[\pi/r(X_1, ..., X_n)] \wedge def(\pi, \psi, r)$ is a formula renaming where r is a new predicate and $free(\phi) = \{X_1, ..., X_n\}$.

$$def(\pi, \psi, r) = \begin{cases} \forall X_1, ..., X_n.[r(X_1, ..., X_n) \supset \phi] & if\ pol(\psi, \pi) = 1 \\ \forall X_1, ..., X_n.[\phi \supset r(X_1, ..., X_n)] & if\ pol(\psi, \pi) = -1 \\ \forall X_1, ..., X_n.[r(X_1, ..., X_n) \Leftrightarrow \phi] & if\ pol(\psi, \pi) = 0 \end{cases}$$

Because of the chosen polarity-dependent definition, which further reduces the quantity of clauses, formula renaming is satisfiability preserving only.

The number of clauses generated by a formula can be calculated with the calculations $p(\psi)$ and $\bar{p}(\psi) = (p(\neg\psi)$ given in table 9.

ψ	$p(\psi)$	$\bar{p}(\psi)$
$\phi_1 \wedge \phi_2$	$p(\phi_1) + p(\phi_2)$	$\bar{p}(\phi_1)\bar{p}(\phi_2)$
$\phi_1 \vee \phi_2$	$p(\phi_1)p(\phi_2)$	$\bar{p}(\phi_1) + \bar{p}(\phi_2)$
$\phi_1 \supset \phi_2$	$\bar{p}(\phi_1)p(\phi_2)$	$p(\phi_1) + \bar{p}(\phi_2)$
$\phi_1 \Leftrightarrow \phi_2$	$p(\phi_1)\bar{p}(\phi_2) + \bar{p}(\phi_1)p(\phi_2)$	$p(\phi_1)p(\phi_2) + \bar{p}(\phi_1)\bar{p}(\phi_2)$
$\forall X.\phi_1$	$p(\phi_1)$	$\bar{p}(\phi_1)$
$\exists X.\phi_1$	$p(\phi_1)$	$\bar{p}(\phi_1)$
$\neg\phi_1$	$\bar{p}(\phi_1)$	$p(\phi_1)$
$p(X_1, ..., X_n)$	1	1

Table 9: Computation of p and $\bar{p}$

With that, the condition for a formula renaming can be stated as:

$$p(\psi) \geq p(\psi[\pi/r(X_1, ..., X_n)]) + p(def(\pi, \psi, r))$$

However, a top-down implementation of p would be exponentially inefficient in computation time. Therefore, two coefficients a_π^ψ and b_π^ψ are presented, which only consider the changed subformulae to compare the number of clauses of both changed and unchanged formulae. Also, intermediate results are stored using dynamic programming principles, and thus avoiding redundant computations. The coefficients a_π^ψ and b_π^ψ are defined in table 10 on the facing page.

Using the newly introduced coefficients the condition can now be formulated as

$$a_\pi^\psi p(\phi) + b_\pi^\psi \bar{p}(\phi) \geq a_\pi^\psi + b_\pi^\psi + p(def(\pi, \psi, r))$$

Since the coefficient a_π^ψ (b_π^ψ) stands for the multiplication of $p(\psi|_\pi)$ ($\bar{p}(\psi|_\pi)$), a_π^ψ (b_π^ψ) is always 0, if $pol(\psi, \pi) = -1$ ($pol(\psi, \pi) = 1$). Thus, the condition can be split into three inequalities:

$$\begin{aligned} a_\pi^\psi p(\phi) &\geq a_\pi^\psi + p(\phi) && if\ pol(\psi, \pi) = 1 \\ b_\pi^\psi \bar{p}(\phi) &\geq b_\pi^\psi + \bar{p}(\phi) && if\ pol(\psi, \pi) = -1 \\ a_\pi^\psi p(\phi) + b_\pi^\psi \bar{p}(\phi) &\geq a_\pi^\psi + b_\pi^\psi + p(\phi) + \bar{p}(\phi) && if\ pol(\psi, \pi) = 0 \end{aligned}$$

π	$\psi\|_\tau$	a_π^ψ	b_π^ψ
$\tau.i$	$\phi_1 \wedge \phi_2$	a_τ^ψ	$b_\tau^\psi \prod_{j\neq i} \bar{p}(\phi_j)$
$\tau.i$	$\phi_1 \vee \phi_2$	$a_\tau^\psi \prod_{j\neq i} p(\phi_j)$	b_τ^ψ
$\tau.1$	$\phi_1 \supset \phi_2$	b_τ^ψ	$a_\tau^\psi p(\phi_2)$
$\tau.2$	$\phi_1 \supset \phi_2$	$a_\tau^\psi \bar{p}(\phi_2)$	b_τ^ψ
$\tau.1$	$\phi_1 \Leftrightarrow \phi_2$	$a_\tau^\psi \bar{p}(\phi_2) + b_\tau^\psi p(\phi_2)$	$a_\tau^\psi p(\phi_2) + b_\tau^\psi \bar{p}(\phi_2)$
$\tau.2$	$\phi_1 \Leftrightarrow \phi_2$	$a_\tau^\psi \bar{p}(\phi_1) + b_\tau^\psi p(\phi_1)$	$a_\tau^\psi p(\phi_1) + b_\tau^\psi \bar{p}(\phi_1)$
$\tau.1$	$\neg\phi_1$	b_τ^ψ	a_τ^ψ
$\tau.1$	$\forall X.\phi_1$	a_τ^ψ	b_τ^ψ
$\tau.1$	$\exists X.\phi_1$	a_τ^ψ	b_τ^ψ
ϵ	ψ	1	0

Table 10: Computation of a_π^ψ and b_π^ψ

$def(\pi, \psi, r)$ can be replaced according to the definition of p and the polarity of ϕ. By reordering and the two abbreviations $p_a = (a_\pi^\psi - 1)(p(\phi) - 1)$ and $p_b = (b_\pi^\psi - 1)(\bar{p}(\phi) - 1)$ the conditions are:

$$\begin{aligned} p_a \geq 1 \qquad & if\ pol(\psi, \pi) = 1 \\ p_b \geq 1 \qquad & if\ pol(\psi, \pi) = -1 \\ p_a \geq 2 \vee p_b \geq 2 \vee (p_a \geq 1 \wedge p_b \geq 1) \qquad & if\ pol(\psi, \pi) = 0 \end{aligned}$$

To check this condition it is sufficient to know whether a_π^ψ, b_π^ψ, $p(\phi)$ and $\bar{p}(\phi)$ are strictly greater than 1, 2 or 3.

$p(\psi) > 1$	
$\exists\pi.\phi[\omega]_\pi$	$\omega = \phi_1 \Leftrightarrow \phi_2$
$\exists\pi.\phi[\omega]_\pi$ and $pol(\phi, \pi) = 1$	$\omega = \phi_1 \wedge \phi_2$
$\exists\pi.\phi[\omega]_\pi$ and $pol(\phi, \pi) = -1$	$\omega = \phi_1 \vee \phi_2$
$\exists\pi.\phi[\omega]_\pi$ and $pol(\phi, \pi) = -1$	$\omega = \phi_1 \supset \phi_2$

Table 11: Computation of $p(\psi) > 1$

ψ	$p(\psi) > 2$
$\phi_1 \wedge \phi_2$	$p(\phi_1) > 1 \vee p(\phi_2) > 1$
$\phi_1 \vee \phi_2$	$p(\phi_i) > 2$ or $p(\phi_1) > 1 \wedge p(\phi_2) > 1$

Table 12: Computation of $p(\psi) > 2$

ψ	$p(\psi) > 3$
$\phi_1 \wedge \phi_2$	$p(\phi_i) > 2$
$\phi_1 \vee \phi_2$	$p(\phi_i) > 3$ or $p(\phi_i) > 2 \wedge p(\phi_{-i}) > 1$

Table 13: Computation of $p(\psi) > 3$

π	$\psi\|_\tau$	$a_\pi^\psi > 1$
$\tau.i$	$\phi_1 \wedge \phi_2$	$a_\tau^\psi > 1$
$\tau.i$	$\phi_1 \vee \phi_2$	$a_\tau^\psi > 1$ or $p(\phi_j) > 1$ for some j

Table 14: Computation of $a_\pi^\psi > 1$

The other cases can be deduced analogously from the given definitions.

This calculation can be applied in higher-order logics as well. To increase the impact, the technique presented in section 4.2 should be performed first.

5 CNF Implementation in Leo-III

This section points out some main features about the implementation of the conjunctive normal form in the project Leo-III.

The code for Leo-III can be found in a GitHub repository[2]. At the time of this thesis's publication, the repository is not yet public, however access can be granted on request.

Test output excerpts of the CNF implementation in Leo-III can be found in the appendix (section 7).

5.1 Data structure

Since with de Bruijn indices a nameless representation of bindings was chosen, formulae are always in α-normal form. Furthermore, the data structure ensures that all formulae are represented in a $\beta\eta$-normal form which is useful for instance in the implementation of section 3.5 *Skolemization*. The formula $\exists.\phi \vee 1$ is considered, where 1 is a de Bruijn index referencing the existential quantifier. Let *sko* be the new Skolem constant, then the quantifier can be removed and the Skolem constant applied to the abstraction of the remaining formula $(\lambda.\phi \vee 1)\, sko$ which after the β-normalization results in the desired $\phi \vee sko$. Because the β-normalization can also take care of the decrementation of the de Bruijn indices, this is a very elegant way to replace quantifiers.

The data structure underlying the representation of terms in Leo-III has a term ordering implemented. Therewith the commutativity law is automatically taken care of. For example in the *simplification* step (section 3.1) a formula $(\phi \wedge \psi) \vee \neg(\psi \wedge \phi)$ is not detected by the rule $\phi \vee \neg\phi \rightarrow \top$, assuming the test for equivalence doesn't consider commutativity itself. With term ordering it's guaranteed that the order in the representation of both subformulae is always the same so that the formula can be simplified.

5.2 Pattern Matching in Scala

Scala offers a very powerful pattern matching capability which can be used to traverse formulae. With those, complex structural requirements can be checked and corresponding rules applied. In listing 1 a code snippet of the Skolemization algorithm implementation can be found. When the pattern of interest occurs, in this case *Exists* in line 2, the transformation of the formula is performed. If the pattern of interest can occur within subformulae, the procedure is performed recursively as in lines 14, 15, 16, 17, 18 or

[2]Leo-III repository: `https://github.com/kiwikern/Leo-III`

19. Since the pattern for application, denoted by $\cdot$, in line 17 also includes $\wedge$ and $\vee$, many different cases can be subsumed if no further distinction is needed. The recursion anchors appear in lines 12 and 13 where literals are left unchanged.

To avoid unnecessary pattern tests, cases that can't occur should be omitted, so that the cases vary in different algorithms. For instance a λ-abstraction can't be a directly quantified formula: $\exists X.(\phi) \implies \phi \not\equiv \lambda X.\psi$.

Listing 1: Skolemization in Leo-III

```
1  def skolemize(formula: Term): Term = formula match {
2      case Exists(ty :::> t) => {
3        val t1 = skolemize(t)
4        val defConst: Set[Term] = formula.symbols.map(x
           => mkAtom(Signature.get(x).key))
5        val freeVar: Seq[Term] = freeVariables(\(ty)(t1)
           ).toSeq
6        val types = freeVar.map(x => x.ty)
7        val skoType = Type.mkFunType(types, ty)
8        val skoVar = mkTermApp(mkAtom(Signature.get.
           freshSkolemVar(skoType)), freeVar)
9        val t2 = replaceExistentialBound(t1, skoVar)
10       decrementDeBruijn(t2)
11     }
12     case s@Symbol(_) => s
13     case s@Bound(_, _) => s
14     case s @@@ t => mkTermApp(skolemize(s), skolemize(
         t))
15     case s @@@@ ty => mkTypeApp(skolemize(s), ty)
16     case Forall(ty :::> s) => Forall(\(ty)(skolemize(s
         )))
17     case f · args => Term.mkApp(skolemize(f), args.map(
         _.fold({ t => Left(skolemize(t))}, Right(_))))
18     case ty :::> s => mkTermAbs(ty, skolemize(s))
19     case TypeLambda(t) => mkTypeAbs(skolemize(t))
20     case x => throw new IllegalArgumentException("
         Unknown Pattern for: " + x.pretty)
21   }
```

6 Conclusion

This thesis demonstrated that a conjunctive normal form can be computed in higher-order logics using very similar algorithms as in a first-order context, both theoretically and practically. Some optimizations applied in FOL are portable as well.

But, it also reveals a wide field for potential new improvements, especially due to the nested structure of arguments in applications (see section 4.2). Since no research regarding higher-order CNF has been published yet, new techniques can be found, and the given ones further evaluated in future works. In particular the modular structure in Leo-III allows for elaborate experiments to find an optimal composition and parametrization. Therewith, the interaction of different mechanisms and boundaries for an effective application can be determined.

The implementation and evaluation of the clause normal form in Leo-III is still an ongoing work. Because the calculus in Leo-III is not yet implemented, other higher-order automated theorem provers like Leo-II can be used for both validation and evaluation. On that note, open questions that can be practically examined are, for example, whether *bounded argument depth* should only be applied for functions of certain types to reduce complexity, or if *formula renaming* loses its positive effect with greater formula depth.

In general, the effect of different improvement methods might also depend on the problem domain and can potentially be improved using machine learning techniques. For that, the TPTP problem library[3] can be used which offers a variety of already categorized problems.

Due to the computation of a clause normal form, proofs can become very large, and thus hard to reconstruct for humans. Yet, an important task for automated theorem provers is to return a comprehensible proof object. Improvements in the CNF algorithm may improve the clarity of such proof objects.

[3]TPTP website: http://www.tptp.org

7 Appendix

Listing 2: Simplification test output in Leo-III

```
Simplicifcation: '<=> · (p · (⊥);$false · (⊥);⊥)' was
   simplified to '~ · (p · (⊥);⊥)'
Simplicifcation: '| · (p · (⊥);$true · (⊥);⊥)' was
   simplified to '$true · (⊥)'
Simplicifcation: '=> · (p · (⊥);$false · (⊥);⊥)' was
   simplified to '~ · (p · (⊥);⊥)'
Simplicifcation: '! · ($o;λ[$o]. (p · (⊥));⊥)' was
   simplified to 'p · (⊥)'
Simplicifcation: '~ · ($false · (⊥);⊥)' was simplified
   to '$true · (⊥)'
Simplicifcation: '| · (p · (⊥);$false · (⊥);⊥)' was
   simplified to 'p · (⊥)'
Simplicifcation: '& · (p · (⊥);~ · (p · (⊥);⊥);⊥)' was
   simplified to '$false · (⊥)'
Simplicifcation: '? · ($o;λ[$o]. (p · (⊥));⊥)' was
   simplified to 'p · (⊥)'
Simplicifcation: '& · (p · (⊥);$false · (⊥);⊥)' was
   simplified to '$false · (⊥)'
Simplicifcation: '=> · (p · (⊥);p · (⊥);⊥)' was
   simplified to '$true · (⊥)'
Simplicifcation: '=> · (p · (⊥);$true · (⊥);⊥)' was
   simplified to '$true · (⊥)'
Simplicifcation: '<=> · (p · (⊥);$true · (⊥);⊥)' was
   simplified to 'p · (⊥)'
Simplicifcation: '| · (p · (⊥);p · (⊥);⊥)' was
   simplified to 'p · (⊥)'
Simplicifcation: '| · (p · (⊥);~ · (p · (⊥);⊥);⊥)' was
   simplified to '$true · (⊥)'
Simplicifcation: '=> · ($false · (⊥);p · (⊥);⊥)' was
   simplified to '$true · (⊥)'
Simplicifcation: '& · (p · (⊥);p · (⊥);⊥)' was
   simplified to 'p · (⊥)'
Simplicifcation: '<=> · (p · (⊥);p · (⊥);⊥)' was
   simplified to '$true · (⊥)'
Simplicifcation: '~ · ($true · (⊥);⊥)' was simplified
   to '$false · (⊥)'
Simplicifcation: '! · ($o;λ[$o]. (<=> · (p · (1 · (⊥);⊥
   );p · (1 · (⊥);⊥);⊥));⊥)' was simplified to '$true ·
    (⊥)'
Simplicifcation: '=> · ($true · (⊥);p · (⊥);⊥)' was
   simplified to 'p · (⊥)'
```

Listing 3: Negation normal form test output in Leo-III

```
Negation:
'~ · (<=> · (r · (⊥);s · (⊥);⊥);⊥)' was normalized to
'& · (| · (~ · (r · (⊥);⊥);~ · (s · (⊥);⊥);⊥);| · (r · (⊥
   );s · (⊥);⊥);⊥)'.
success
Negation:
'~ · (? · ($o;λ[$o}]. (r · (⊥));⊥);⊥)' was normalized
   to
'! · ($o;λ[$o}]. (~ · (r · (⊥);⊥));⊥)'.
success
Negation:
'=> · (r · (⊥);s · (⊥);⊥)' was normalized to
'| · (~ · (r · (⊥);⊥);s · (⊥);⊥)'.
success
Negation:
'<=> · (r · (⊥);s · (⊥);⊥)' was normalized to
'& · (| · (~ · (r · (⊥);⊥);s · (⊥);⊥);| · (~ · (s · (⊥);⊥
   );r · (⊥);⊥);⊥)'.
success
Negation:
'~ · (! · ($o;λ[$o}]. (r · (⊥));⊥);⊥)' was normalized
   to
'? · ($o;λ[$o}]. (~ · (r · (⊥);⊥));⊥)'.
success
Negation:
'~ · (~ · (r · (⊥);⊥);⊥)' was normalized to
'r · (⊥)'.
success
Negation:
'~ · (| · (r · (⊥);s · (⊥);⊥);⊥)' was normalized to
'& · (~ · (r · (⊥);⊥);~ · (s · (⊥);⊥);⊥)'.
success
Negation:
'~ · (& · (r · (⊥);s · (⊥);⊥);⊥)' was normalized to
'| · (~ · (r · (⊥);⊥);~ · (s · (⊥);⊥);⊥)'.
success
```

Listing 4: Miniscoping test output in Leo-III

```
##############################
## miniscoping 1
##############################
Formula :'! · ($o;λ[$o}]. (? · ($o;λ[$o}]. (! · ($o;λ[$o
    }]. (& · (r · (3 · (⊥);3 · (⊥);⊥);| · (| · (p · (2 · (
    ⊥);⊥);r · (3 · (⊥);2 · (⊥);⊥);⊥);q · (3 · (⊥);⊥);⊥)
    ;⊥));⊥));⊥));⊥)'
was normalized to: '& · (! · ($o;λ[$o}]. (r · (1 · (⊥);1
     · (⊥);⊥));⊥);! · ($o;λ[$o}]. (| · (| · (? · ($o;λ[$o
    }]. (p · (1 · (⊥);⊥));⊥);? · ($o;λ[$o}]. (r · (2 · (⊥
    );1 · (⊥);⊥));⊥);⊥);q · (1 · (⊥);⊥);⊥));⊥);⊥)'
Success
##############################
## miniscoping 2
##############################
Formula :'! · ($o;λ[$o}]. (& · (1 · (⊥);2 · (⊥);⊥));⊥)'
was normalized to: '& · (! · ($o;λ[$o}]. (1 · (⊥));⊥);1
    · (⊥);⊥)'
Success
##############################
## miniscoping 3 - remove unnecessary quantifiers
##############################
Formula :'! · ($o;λ[$o}]. (! · ($o;λ[$o}]. (? · ($o;λ[$o
    }]. (! · ($o;λ[$o}]. (! · ($o;λ[$o}]. (3 · (⊥));⊥));⊥
    ));⊥));⊥));⊥)'
was normalized to: '? · ($o;λ[$o}]. (1 · (⊥));⊥)'
Success
##############################
## miniscoping 4
##############################
Formula :'! · ($o;λ[$o}]. (! · ($o;λ[$o}]. (r · (2 · (⊥)
    ;2 · (⊥);⊥));⊥));⊥)'
was normalized to: '! · ($o;λ[$o}]. (r · (1 · (⊥);1 · (⊥
    );⊥));⊥)'
Success
##############################
## miniscoping 5
##############################
Formula :'! · ($o;λ[$o}]. (| · (| · (p · (2 · (⊥);⊥);r ·
    (2 · (⊥);1 · (⊥);⊥);⊥);q · (1 · (⊥);⊥);⊥));⊥)'
was normalized to: '! · ($o;λ[$o}]. (| · (| · (p · (2 · (
    ⊥);⊥);r · (2 · (⊥);1 · (⊥);⊥);⊥);q · (1 · (⊥);⊥);⊥)
    );⊥)'
Success
```

Listing 5: Skolemization test output in Leo-III

```
################################
## SYN993^1.p
################################
Formula "| · (? · ($i;λ[$i}]. (! · ($i;λ[$i}]. (100 · (2 ·
    (⊥);1 · (⊥);⊥));⊥));⊥);? · ($i;λ[$i}]. (! · ($i;λ[$
   i}]. (~ · (100 · (1 · (⊥);2 · (⊥);⊥);⊥));⊥));⊥);⊥)"
was skolemized to: "| · (! · ($i;λ[$i}]. (99 · (sk1 · (99
    · (⊥);⊥);1 · (⊥);⊥));⊥);! · ($i;λ[$i}]. (~ · (99 ·
   (1 · (⊥);sk2 · (99 · (⊥);⊥);⊥);⊥));⊥);⊥)".
success
################################
## SYN994^1.p
################################
Formula "? · ($i;λ[$i}]. (! · ($i;λ[$i}]. (! · ($i;λ[$i
   }]. (| · (100 · (2 · (⊥);1 · (⊥);⊥);~ · (100 · (2 · (⊥
   );3 · (⊥);⊥);⊥);⊥));⊥));⊥));⊥)"
was skolemized to: "! · ($i;λ[$i}]. (! · ($i;λ[$i}]. (| ·
    (99 · (2 · (⊥);1 · (⊥);⊥);~ · (99 · (2 · (⊥);sk1 ·
   (99 · (⊥);⊥);⊥);⊥);⊥));⊥));⊥)".
success
################################
## SYN996^1.p
################################
Formula "=> · (! · ($i;λ[$i}]. (? · ($i;λ[$i}]. (100 · (2
    · (⊥);1 · (⊥);⊥));⊥));⊥);? · ($i -> $i;λ[$i -> $i
   }]. (! · ($i;λ[$i}]. (100 · (1 · (⊥);2 · (1 · (⊥);⊥);
   ⊥));⊥));⊥);⊥)"
was skolemized to: "=> · (! · ($i;λ[$i}]. (99 · (1 · (⊥)
   ;sk1 · (99 · (⊥);1 · (⊥);⊥);⊥));⊥);! · ($i;λ[$i}].
   (99 · (1 · (⊥);sk2 · (99 · (⊥);1 · (⊥);⊥);⊥));⊥);⊥)"
   .
success
################################
## SYN997^1.p
################################
Formula "? · (($i -> $o) -> $i;λ[($i -> $o) -> $i}]. (!
   · ($i -> $o;λ[$i -> $o}]. (=> · (? · ($i;λ[$i}]. (2 ·
   (1 · (⊥);⊥));⊥);1 · (2 · (1 · (⊥);⊥);⊥);⊥));⊥));⊥)"
was skolemized to: "! · ($i -> $o;λ[$i -> $o}]. (=> · (1
    · (sk1 · (1 · (⊥);⊥);⊥);1 · (sk2 · (1 · (⊥);⊥);⊥);⊥)
   );⊥)".
success
```

Listing 6: Prenex normal form test output in Leo-III

```
################################
## Prenex 1
################################
The Term '& · (! · ($o;λ[$o]. (1 · (⊥));⊥);sk1 · (⊥);⊥)
    '
 was normalized to '! · ($o;λ[$o]. (& · (1 · (⊥);sk1 ·
    (⊥);⊥));⊥)'.
################################
## Prenex 2
################################
The Term '& · (sk1 · (⊥);! · ($o;λ[$o]. (1 · (⊥));⊥);⊥)
    '
 was normalized to '! · ($o;λ[$o]. (& · (sk1 · (⊥);1 ·
    (⊥);⊥));⊥)'.
################################
## Prenex 3
################################
The Term '& · (! · ($o;λ[$o]. (1 · (⊥));⊥);! · ($o;λ[$o
   }]. (1 · (⊥));⊥);⊥)'
 was normalized to '! · ($o;λ[$o]. (! · ($o;λ[$o]. (& ·
     (2 · (⊥);1 · (⊥);⊥));⊥));⊥)'.
```

Listing 7: Clause normal form test output in Leo-III

```
################################
## ClauseForm 1
################################
Term '1 · (⊥)'
 was normalized to '[[1 · (⊥)] = T]  '
################################
## ClauseForm 2
################################
Term '~ · (p · (⊥);⊥)'
 was normalized to '[[p · (⊥)] = F]  '
################################
## ClauseForm 3
################################
Term '| · (1 · (⊥);| · (p · (⊥);q · (⊥);⊥);⊥)'
 was normalized to '[[1 · (⊥)] = T , [p · (⊥)] = T , [q
     · (⊥)] = T]  '
################################
## ClauseForm 4
################################
Term '& · (& · (1 · (⊥);| · (q · (⊥);p · (⊥);⊥);⊥);| · (p
    · (⊥);q · (⊥);⊥);⊥)'
 was normalized to '[[1 · (⊥)] = T]  [[q · (⊥)] = T , [
    p · (⊥)] = T]  [[p · (⊥)] = T , [q · (⊥)] = T]  '
```

Bibliography

[And72] Peter B Andrews. General models, descriptions, and choice in type theory. *Journal of symbolic logic*, pages 385–394, 1972. 1.2

[And81] Peter B. Andrews. Theorem proving via general matings. *J. ACM*, 28(2):193–214, April 1981. 1.4

[AvK08] Aristoteles and Julius H von Kirchmann. *Organon: Deutsche Übersetzung von Julius Heinrich von Kirchmann (1876):[Klassiker]*. GRIN-Verlag, 2008. 1.1

[BBK04] Christoph Benzmüller, Chad E Brown, and Michael Kohlhase. Higher-order semantics and extensionality. *The Journal of Symbolic Logic*, 69(04):1027–1088, 2004. 2, 4.2

[Ben13] Christoph Benzmüller. Automating quantified conditional logics in HOL. In Francesca Rossi, editor, *23rd International Joint Conference on Artificial Intelligence (IJCAI-13)*, pages 746–753, Beijing, China, 2013. 1.2

[Ben15] Christoph Benzmüller. Higher-order automated theorem provers. In David Delahaye and Bruno Woltzenlogel Paleo, editors, *All about Proofs, Proof for All*, Mathematical Logic and Foundations, pages 171–214. College Publications, London, UK, 2015. 2, 2

[BM14] Christoph Benzmüller and Dale Miller. Automation of higher-order logic. In Dov M. Gabbay, Jörg H. Siekmann, , and John Woods, editors, *Handbook of the History of Logic, Volume 9 — Computational Logic*, pages 215–254. North Holland, Elsevier, 2014. 1.2, 2

[BRS08] Christoph Benzmüller, Florian Rabe, and Geoff Sutcliffe. THF0 – the core of the TPTP language for classical higher-order logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *LNCS*, pages 491–506. Springer, 2008. 1.2

[BTPF08] Christoph Benzmüller, Frank Theiss, Lawrence Paulson, and Arnaud Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *LNCS*, pages 162–170. Springer, 2008. 1.3

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(02):56–68, 1940. 1.1, 1.2, 2

[CP03] Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003. 2

[DB72] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972. 2, 3.4, 3.6

[Fre64] Gottlob Frege. *Begriffsschrift und andere Aufsätze*. Georg Olms Verlag, 1964. 1.1

[Göd31] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931. 1.1, 1.2

[Göd95] Kurt Gödel. Ontological proof. *Kurt Gödel collected works*, 3:403–404, 1995. 1.2

[Hen50] Leon Henkin. Completeness in the theory of types. *The Journal of Symbolic Logic*, 15(02):81–91, 1950. 1.1, 1.2

[Hue75] Gerard P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975. 4.2

[Lei65] Gottfried Wilhelm Leibniz. Characteristica universalis. *Die philosophischen Schriften von Gottfried Wilhelm Leibniz. Hrsg. v. CJ Gerhardt. Nachdruck. Hildesheim*, pages 184–189, 1965. 1.1

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002. 1.3

[NW01] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. *Handbook of automated reasoning*, 1:335–367, 2001. 3, 4.4

[oP15] Stanford Encylcopedia of Philosophy. Church's type theory, http://plato.stanford.edu/entries/type-theory-church/, 03-05 2015. 1.1

[RV01] Alexandre Riazanov and Andrei Voronkov. Splitting without backtracking. In *IJCAI*, pages 611–617. Citeseer, 2001. 4.1

[Sch04] Stephan Schulz. System description: E 0.81. In *Automated Reasoning*, pages 223–228. Springer, 2004. 1.3

[Ste14] Alexander Steen. *Efficient Data Structures for Automated Theorem Proving in Expressive Higher-Order Logics*. PhD thesis, fu-berlin, 2014. 1.3

[Sut09] Geoff Sutcliffe. The tptp problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009. 1.2, 1.3

[Wis14] Max Wisniewski. *Agent-based Blackboard Architecture for a Higher-Order Theorem Prover*. PhD thesis, fu-berlin, 2014. 1.3

[WR12] Alfred North Whitehead and Bertrand Russell. *Principia mathematica*, volume 2. University Press, 1912. 1.1

[WSB14] Max Wisnieski, Alexander Steen, and Christoph Benzmüller. The Leo-III project. In Alexander Bolotov and Manfred Kerber, editors, *Joint Automated Reasoning Workshop and Deduktionstreffen*, page 38, 2014. 1.3

List of Tables