New Techniques in Clausal Form Generation

Giles Reger[1], Martin Suda[2], and Andrei Voronkov[1,3,4,*]

[1] University of Manchester, Manchester, UK
[2] Institute for Information Systems, Vienna University of Technology, Austria
[3] Chalmers University of Technology, Gothenburg, Sweden
[4] EasyChair

Abstract

In automated reasoning it is common that first-order formulas need to be translated into clausal normal form for proof search. The structure of this normal form can have a large impact on the performance of first-order theorem provers, influencing whether a proof can be found and how quickly. It is common folklore that transformations should ideally minimise both the size of the generated clause set and extensions to the signature. This paper introduces a new top-down approach to clausal form generation for first-order formulas that aims to achieve this goal in a new way. The main advantage of this approach over existing bottom-up techniques is that more contextual information is available at points where decisions such as subformula-naming and Skolemisation occur. Experimental results show that our implementation of the transformation in Vampire can lead to clausal forms which are smaller and better suited to proof search.

1 Introduction

Many applications of automated reasoning (verification, program analysis) produce problems in the form of first-order formulas. However, theorem provers often require problems in *clausal normal form* (*CNF* for short) for proof search. The process of generating a clausal form from a first-order formula is well-known. It is also well-known that the transformation process can have a large impact on the subsequent proof search. It is common folklore that transformations should ideally minimise both the size of the generated clause set and extensions to the signature. However, it is not possible to determine the optimal clausal form and these measures are heuristic.

Clausal form transformations typically [5] take the form of iterative passes through the formula tree with each pass moving the formula one step closer towards clausal form. As discussed later, these steps involve expanding equivalences, naming subformulas and Skolemisation. Optimisations and improvements [1, 3] have been proposed for each step, but the general approach of making iterative passes over the formula tree is the same.

We introduce a new algorithm for clausal form transformation called VCNF. Our approach differs from existing work as it employs a single *top-down* traversal, i.e., it steps through the formula tree once producing the clausal form as it goes. The above stages are therefore no longer

*The work of this author is supported by the Wallenberg Foundation.

independent, allowing optimisations to use context from what were previously independent stages. For instance, equivalence expansion and Skolemisation are typically separate steps, but it is possible to use information from equivalence expansion to introduce fewer Skolem functions, leading to a smaller signature. Another advantage is an easy detection of intermediate tautologies, which are discarded on the fly. Our approach thus maintains a more faithful count of subformula occurrences, on which the decision whether to name a subformula is based.

The algorithm has been implemented in the Vampire theorem prover [2]. Sections 3, 4 and 6 of this paper discuss examples and experimental results showing that the new approach can lead to more friendly clausal forms than existing techniques.

The main contributions of this paper are:

1. a new top-down algorithm for clausal form transformation (Section 3),
2. an extension for naming subformulas (Section 4),
3. an extension for detecting and eliminating *tautologies* (Section 6),
4. an extension for inlining context at intermediate steps in the transformation (Section 7).

In addition, we describe a previously unpublished algorithm for CNF transformation based on a *naming threshold* parameter, and a notion of *equivalence negation normal form*, which simplifies presentation of CNF transformation algorithms. Both the naming threshold and equivalence negation normal form were implemented in several previous versions of Vampire.

2 Previous Work on Clausal Form Generation

We review existing work on clausal form generation. This will be given within the context of the previous clausal form generation approach implemented in the Vampire theorem prover for two reasons: (i) we must settle on one variation of the transformation, (ii) understanding how Vampire currently implements this process is necessary to interpret experimental results later. The discussion will include references to other optimisations and previous work not covered by the Vampire approach where necessary.

2.1 Preliminaries

Our setting is that of first-order predicate logic with equality.

A signature Σ is a set of *predicate* and *function* symbols with associated arities. A *term* is of the form $f(t_1, \ldots, t_n)$, c or x where f is a *function symbol* of arity $n \geq 1$, $t_1, \ldots, t_n$ are terms, c is a zero arity function symbol (i.e. a constant) and x is a variable. An *atom* is of the form $p(t_1, \ldots, t_n)$, q or $t_1 \simeq t_2$ where p is a *predicate symbol* of arity n, $t_1, \ldots, t_n$ are terms, q is a zero arity predicate symbol and $\simeq$ is the *equality symbol*. A literal is an atom or its negation.

A *formula* is of the form $\varphi_1 \wedge \ldots \wedge \varphi_n$, $\varphi_1 \vee \ldots \vee \varphi_n$, $\varphi_1 \rightarrow \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$, $\varphi_1 \otimes \varphi_2$, $\neg\varphi_1$, $\exists x.\varphi_1$, $\forall x.\varphi_1$, $\bot$, $\top$, or l where φ_i are formulas, x is a variable and l is a literal. Note that we treat conjunction and disjunction as n-ary operators; we assume that formulas are kept in *flattened form*, e.g. $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3$ is always represented as $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$. Furthermore, we assume that usage of $\top$ and $\bot$ is simplified immediately. Let $\mathsf{fvars}(\varphi)$ be the *free variables* of formula φ, i.e. those variables in φ with an occurrence not bound by a quantifier. We will sometimes write $\varphi[x_1, \ldots, x_n]$ to mean that $x_1, \ldots, x_n = \mathsf{fvars}(\varphi)$ and $\mathbf{x}$ as shorthand for $x_1, \ldots, x_n$.

A *position* is a word over natural numbers. The positions $\mathsf{pos}(\varphi)$ of formula φ are defined as follows. The empty position ϵ is in $pos(\varphi)$. If $\varphi = \varphi_1 \circ \ldots \circ \varphi_n$ for some operator $\circ$ and $p \in \mathsf{pos}(\varphi_i)$ then $i.p \in \mathsf{pos}(\varphi)$. Let $\varphi\mid_\epsilon = \varphi$ and $\varphi\mid_{i.p} = \varphi_i\mid_p$ where $\varphi = \varphi_1 \circ \ldots \circ \varphi_n$. The *polarity* of a subformula $\varphi\mid_\pi$ is given by $\mathsf{pol}(\varphi, \pi)$, defined as $\mathsf{pol}(\varphi, \epsilon) = 1$; $\mathsf{pol}(\varphi, \pi.i) = \mathsf{pol}(\varphi, \pi)$ if the top-level symbol of $\varphi\mid_\pi$ is a $\vee, \wedge, \forall$, or $\exists$, or the right of $\rightarrow$ $(i = 2)$; $\mathsf{pol}(\varphi, \pi.i) = -\mathsf{pol}(\varphi, \pi)$ if the top-level symbol of $\varphi\mid_\pi$ is $\neg$ or the left of $\rightarrow$ $(i=1)$; and $\mathsf{pol}(\varphi,\pi.i)=0$ if the top-level symbol of $\varphi\mid_\pi$ is $\leftrightarrow$ or $\otimes$.

A *substitution* is any expression σ of the form $\{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$, where $n \geq 0$, x_i are distinct variables, and t_i are terms. $E\sigma$ is the expression obtained from E by the simultaneous replacement of each occurrence of x_i in E by t_i. By an expression we here mean a term, an atom, a literal, or a formula. An expression is *ground* if it contains no variables.

A *clause* is a disjunction of literals $L_1 \vee \ldots \vee L_n$ for $n \geq 0$. We disregard the order of literals and treat a clause as a multiset. Variables in clauses are considered to be universally quantified.

2.2 Transformation Steps

The standard clausal form transformation makes the following steps on each formula to produce a set of clauses.

1. Rectification
2. Conversion to ENNF
3. Subformula Naming
4. Conversion to NNF
5. Skolemization
6. Transformation into clauses

This is similar to that of [1]. We discuss the straightforward steps here and the more involved steps of subformula naming and Skolemisation in the next two subsections.

Rectification. Bound variables are renamed to ensure that (i) no variable is free and bound, and (ii) for every variable x there is at most one occurrence of $\forall x$ or $\exists x$. This is not necessary in itself but can simplify later steps and their implementation. The formula size is not changed.

Conversion to ENNF. A formula is in *equivalence negation normal form* (ENNF) if it does not contain $\rightarrow$ and negations are only applied to atoms. Notably the equivalence $\leftrightarrow$ and non-equivalence (xor) $\otimes$ connectives are not expanded at this point. This allows the subformula naming technique to name subformulas. Conversion to ENNF can only give a linear increase in the size of the formula. Note that the notion of ENNF was introduced and used in several previous versions of Vampire, as a convenient intermediate form, which is similar to NNF, but requires only a linear time transformation.

Conversion to NNF. A formula is in *negation normal form* (NNF) if it does not contain $\rightarrow$, $\leftrightarrow$, or $\otimes$ and negations are only applied to atoms. Transforming a formula from ENNF to NNF can result in an exponential increase in the size of the formula, as arguments to the removed equivalences (and non-equivalences) need to be copied. Subformula naming aims to prevent the ensuing increase in the formula size.

Transformation into clauses. At this point the formula contains only conjunctions and disjunctions over literals and can be transformed into a set of clauses by applying distributivity rules. Exhaustive distribution may also increase the formula size exponentially. Again, this is something that can be prevented with subformula naming.

2.3 Subformula Naming

We recall the standard motivation for subformula naming and then describe the current state-of-the-art algorithm.

Table 1: Calculating the number of clauses generated from a formula.

φ	$\alpha_+(\varphi)$	$\alpha_-(\varphi)$
$\varphi_1 \wedge \ldots \wedge \varphi_n$	$\Sigma_{i=1}^n \ \alpha_+(\varphi_i)$	$\Pi_{i=0}^n \ \alpha_-(\varphi_i)$
$\varphi_1 \vee \ldots \vee \varphi_n$	$\Pi_{i=1}^n \ \alpha_+(\varphi_i)$	$\Sigma_{i=0}^n \ \alpha_-(\varphi_i)$
$\varphi_1 \rightarrow \varphi_2$	$\alpha_-(\varphi_1)\alpha_+(\varphi_2)$	$\alpha_+(\varphi_1) + \alpha_-(\varphi_2)$
$\varphi_1 \leftrightarrow \varphi_2, \varphi_1 \otimes \varphi_2$	$\alpha_+(\varphi_1)\alpha_-(\varphi_2) + \alpha_-(\varphi_1)\alpha_+(\varphi_2)$	$\alpha_+(\varphi_1)\alpha_+(\varphi_2) + \alpha_-(\varphi_1)\alpha_-(\varphi_2)$
$\exists x.\varphi_1, \forall x.\varphi_1$	$\alpha_+(\varphi_1)$	$\alpha_-(\varphi_1)$
$\neg\varphi_1$	$\alpha_-(\varphi_1)$	$\alpha_+(\varphi_1)$

Motivation. Consider the following formula in equivalence negation normal form

$$\varphi_1[x_1, \ldots x_k] \vee \varphi_2.$$

If the transformation of $\varphi_1[x_1, \ldots x_k]$ and φ_2 produce n and m clauses respectively then the overall formula will produce nm clauses. In general, alternating disjunctions and conjunctions can lead to an exponential number of clauses due to the repeated application of distributivity. The standard solution to this problem is *subformula naming* where a fresh predicate symbol p is used to name a subformula and take its place in the formula. The above formula would be replaced by

$$(p(x_1, \ldots, x_k) \vee \varphi_2) \wedge (p(x_1, \ldots, x_k) \rightarrow \varphi_1[x_1, \ldots, x_k])$$

where p is a fresh predicate symbol with arity k. In this case an implication is used to introduce $p(x_1, \ldots, x_k)$ as the subformula φ_1 has positive polarity. The transformation of the resulting formula will produce $n + m$ clauses, avoiding the worst-case exponential explosion.

Polarity-Aware Name Introduction. To name $\varphi\mid_\pi$ where $\mathsf{fvars}(\varphi\mid_\pi) = \mathbf{x}$ we replace $\varphi\mid_\pi$ by $p(\mathbf{x})$ for a fresh predicate symbol p and add the definition $\mathsf{def}(\varphi, \pi, p)$ defined as

$$\mathsf{def}(\varphi, \pi, p) = \begin{cases} \forall \mathbf{x}.(p(\mathbf{x}) \rightarrow \varphi\mid_\pi) & \textit{if } \mathsf{pol}(\varphi, \pi) = 1 \\ \forall \mathbf{x}.(\varphi\mid_\pi \rightarrow p(\mathbf{x})) & \textit{if } \mathsf{pol}(\varphi, \pi) = -1 \\ \forall \mathbf{x}.(p(\mathbf{x}) \leftrightarrow \varphi\mid_\pi) & \textit{if } \mathsf{pol}(\varphi, \pi) = 0 \end{cases}$$

Making definitions polarity-aware reduces the number of clauses introduced and remains satisfiability preserving (see later).

When to Name. As the goal is to reduce the number of produced clauses, it seems sensible to only name a subformula if this will result in fewer generated clauses. Table 1 describes how to compute $\alpha_+(\varphi)$ the number of clauses generated from a positive occurrence of the formula φ (assuming the polarity-aware expansion of $\leftrightarrow$ and $\otimes$ described later). Ideally, one would compute the number of clauses introduced with and without the naming of a subformula. However, this is not efficient; a naive implementation of α is exponential and its value grows exponentially. In the following we sketch two approaches to naming that aim to avoid this issue.

The FLOTTER Approach. As described in [3], when computing the difference in number of clauses it is only necessary to consider the part of φ that has changed. This leads to a reformulation of the rules in Table 1 that computes the coefficients for $\alpha_\star(\varphi\mid_\pi)$ in $\alpha_\star(\varphi)$ for $\star \in \{+, -\}$. Then it is computed that, for instance, if $\mathsf{pol}(\varphi, \pi) = 1$, $\alpha_+(\varphi\mid_\pi) > 1$ and the coefficient of $\alpha_+(\varphi\mid_\pi)$ in $\alpha_+(\varphi)$ is > 1 then $\varphi\mid_\pi$ should be named. Similar linearly-checkable boolean conditions are introduced for the other cases. The result is an approach that computes the above criteria in linear time.

The VAMPIRE **Approach.** Instead of comparing the number of clauses produced with and without naming, VAMPIRE uses a *naming threshold* n_{thresh} and names a formula if it would produce more than n_{thresh} clauses. Naming is then applied in a bottom-up fashion. Starting at the leaves of the formula, the clause count as per Table 1 is passed to the parent position if this count is less than the threshold. Otherwise, subformulas are named (starting with the subformula with the largest α) until a value below the threshold can be passed up.

2.4 Skolemization

To obtain a clausal normal form of a formula, it is necessary to remove existential quantifications. This is achieved by a process called Skolemisation.

Standard Skolemization. The standard operation is described as follows

$$\begin{array}{lcl} \forall x.\varphi[y_1,\ldots,y_n,x] & \Rightarrow & \varphi[y_1,\ldots,y_n,x] \\ \exists x.\varphi[y_1,\ldots,y_n,x] & \Rightarrow & \varphi[y_1,\ldots,y_n,x]\{x \mapsto f(y_1,\ldots,y_n)\} \end{array}$$

i.e. universal quantifications are dropped and existential quantifications are replaced by fresh Skolem functions.

Optimisation. A well-known optimisation for Skolemization is *mini-scoping* or *anti-prenexing* where quantifiers are moved inwards as far as possible. The idea is to reduce the arity of introduced Skolem functions. Consider the formula

$$\forall x.\exists y.(p(x) \vee q(y))$$

the standard Skolemization approach would produce $p(x) \vee q(f(x))$ but pushing the existential quantification in to get $\forall x.(p(x) \vee \exists y.q(y))$ would result in $p(x) \vee q(a)$. However, it may sometimes be optimal to pull quantifiers outwards. This is called *maxiscoping* and can be used to reduce the number of Skolem functions introduced. Consider the formula

$$(\exists x.p(x)) \vee (\exists y.q(y))$$

the standard Skolemization approach would produce $p(a) \vee q(b)$ but pulling the existential quantifications out to get $\exists x.(p(x) \vee q(x))$ would result in $p(a) \vee q(a)$. Therefore, it is not straightforward to say one is better than the other or when they should be applied. VAMPIRE does not currently apply miniscoping or maxiscoping.

2.5 Equisatisfiability of Transformation

It is a standard result that the transformations discussed here are satisfiability preserving. But we note that the result is somewhat stronger. They are model-preserving on the original signature. If $\mathcal{I}$ is a model of φ before either Skolemization or subformula naming is applied and φ' is the resulting formula after the transformation, then $\mathcal{I}$ can be extended to a model of φ' by only adding appropriate interpretations for the new symbols.

3 VCNF: A Top-Down Clausification Algorithm

The section describes a top-down clausification algorithm that produces a finite set of clauses from a first-order formula in *equivalence negation normal form* (ENNF).

$$
\begin{array}{lll}
\textit{Given} & \langle\{\{D^i,\varphi^{\mathsf{t}}\}_{\sigma_i},\{D^j,\varphi^{\mathsf{f}}\}_{\sigma_j},D^k_{\sigma_k}\},\varphi.\Delta\rangle & \\[1ex]
\textit{if } \varphi = l & \Rightarrow & \langle\{\{D^i,\varphi^{\mathsf{t}}\}_{\sigma_i},\{D^j,\varphi^{\mathsf{f}}\}_{\sigma_j},D^k_{\sigma_k}\},\Delta\rangle \\
\textit{if } \varphi = \neg\varphi_1 & \Rightarrow & \langle\{\{D^i,\varphi_1^{\mathsf{f}}\}_{\sigma_i},\{D^j,\varphi_1^{\mathsf{t}}\}_{\sigma_j},D^k_{\sigma_k}\},\Delta.\varphi_1\rangle \\
\textit{if } \varphi = \varphi_1 \vee \varphi_2 & \Rightarrow & \langle\{\{D^i,\varphi_1^{\mathsf{t}},\varphi_2^{\mathsf{t}}\}_{\sigma_i},\{D^j,\varphi_1^{\mathsf{f}}\}_{\sigma_j},\{D^j,\varphi_2^{\mathsf{f}}\}_{\sigma_j},D^k_{\sigma_k}\},\Delta.\varphi_1.\varphi_2\rangle \\
\textit{if } \varphi = \varphi_1 \wedge \varphi_2 & \Rightarrow & \langle\{\{D^i,\varphi_1^{\mathsf{t}}\}_{\sigma_i},\{D^i,\varphi_2^{\mathsf{t}}\}_{\sigma_i},\{D^j,\varphi_1^{\mathsf{f}},\varphi_2^{\mathsf{f}}\}_{\sigma_j},D^k_{\sigma_k}\},\Delta.\varphi_1.\varphi_2\rangle \\
\textit{if } \varphi = \varphi_1 \leftrightarrow \varphi_2 & \Rightarrow & \langle\{\{D^i,\varphi_1^{\mathsf{f}},\varphi_2^{\mathsf{t}}\}_{\sigma_i},\{D^i,\varphi_1^{\mathsf{t}},\varphi_2^{\mathsf{f}}\}_{\sigma_i}, \\
 & & \quad \{D^j,\varphi_1^{\mathsf{f}},\varphi_2^{\mathsf{f}}\}_{\sigma_j},\{D^j,\varphi_1^{\mathsf{t}},\varphi_2^{\mathsf{t}}\}_{\sigma_j},D^k_{\sigma_k}\},\Delta.\varphi_1.\varphi_2\rangle \\
\textit{if } \varphi = \varphi_1 \otimes \varphi_2 & \Rightarrow & \langle\{\{D^i,\varphi_1^{\mathsf{f}},\varphi_2^{\mathsf{f}}\}_{\sigma_i},\{D^i,\varphi_1^{\mathsf{t}},\varphi_2^{\mathsf{t}}\}_{\sigma_i}, \\
 & & \quad \{D^j,\varphi_1^{\mathsf{f}},\varphi_2^{\mathsf{t}}\}_{\sigma_j},\{D^j,\varphi_1^{\mathsf{t}},\varphi_2^{\mathsf{f}}\}_{\sigma_j},D^k_{\sigma_k}\},\Delta.\varphi_1.\varphi_2\rangle \\
\textit{if } \varphi = \forall x.\varphi_1 & \Rightarrow & \langle\{\{D^i,\varphi_1^{\mathsf{t}}\}_{\sigma_i},\{D^j,\varphi_1^{\mathsf{f}}\}_{\sigma_j\cup\{x\mapsto f_{\varphi\sigma_j}(z_1,\dots,z_m)\}},D^k_{\sigma_k}\},\Delta.\varphi_1\rangle \\
 & & \textit{where } \{z_1,\dots,z_m\} = \textit{fvars}(\varphi\sigma_j) \textit{ and } f_{\varphi\sigma_j} \textit{ fresh with arity } m \\
\textit{if } \varphi = \exists x.\varphi_1[y_1,\dots,y_n] & \Rightarrow & \langle\{\{D^i,\varphi_1^{\mathsf{t}}\}_{\sigma_i\cup\{x\mapsto f_{\varphi\sigma_i}(z_1,\dots,z_m)\}},\{D^j,\varphi_1^{\mathsf{f}}\}_{\sigma_j},D^k_{\sigma_k}\},\Delta.\varphi_1\rangle \\
 & & \textit{where } \{z_1,\dots,z_m\} = \textit{fvars}(\varphi\sigma_i) \textit{ and } f_{\varphi\sigma_i} \textit{ fresh with arity } m
\end{array}
$$

Figure 1: VCNF rules.

3.1 The Basic Algorithm

Sequents. A sign is either t or f. A *signed formula* is a pair consisting of a formula φ and a sign $\star$, denoted by $\varphi^\star$. The signed formula φ^{t} (resp. φ^{f}) means that φ is true (resp. false). We use the mapping form from signed formulas to formulas defined as follows: $\mathsf{form}(\varphi^{\mathsf{t}}) = \varphi$ and $\mathsf{form}(\varphi^{\mathsf{f}}) = \neg\varphi$. We call a *sequent* a finite set of signed formulas. We say that a sequent $S_1,\dots,S_n$ is true in an interpretation $\mathcal{I}$ if the universal closure of the formula $\mathsf{form}(S_1) \vee \dots \vee \mathsf{form}(S_n)$ is true in $\mathcal{I}$. Note that if $S_1,\dots S_n$ are signed atoms then $\mathsf{form}(S_1) \vee \dots \vee \mathsf{form}(S_n)$ is a clause.

The VCNF algorithm works with finite sets of sequents. While the algorithm is working, we keep constructing substitutions to be applied to existing (signed) formulas. It is convenient for us to collect these substitutions without applying them right away. For this reason, instead of a sequent $D\sigma$, where σ is a substitution, we use pairs D_σ consisting of a sequent D and a substitution σ. We (slightly informally) also refer to such pairs as sequents.

Rules. The VCNF algorithm is captured by the rules in Figure 1 which rewrite a *configuration* $\langle\Gamma,\Delta\rangle$ where Γ is a finite set of sequents and Δ is a sequence of formulas. In Figure 1, we use the notation $\{D^i,\varphi^{\mathsf{t}}\}$ to select all sequents i that contain φ with the positive sign (and analogously for the negative sign). The following transformation should then apply to all such sequents.

The rewriting is driven by the sequence Δ, the left-most formula of which determines which rewriting to apply next. To translate a formula φ into clausal normal form one should begin with $\langle\Gamma,\Delta\rangle = \langle\{\{\varphi^{\mathsf{t}}\}_\epsilon\},\langle\varphi\rangle\rangle$, where ϵ is the empty substitution, and apply the rules until Δ is empty. (We overload the notation and later also denote the empty sequence by ϵ.)

The rule for literals is straightforward; the literal is kept untransformed. In the rest of the paper we will skip this step in examples. For negation the signs are simply swapped. The rules for disjunction and conjunction follow the standard transformation and are dual in polarity, i.e. for positive disjunction the formula is expanded within the same sequent (which itself stands for a disjunction) and for positive conjunction two new sequents must be created (they distribute). An extension of the rules to cover conjunctions and disjunctions of arbitrary arity is straightforward and left out from Figure 1 for the sake of clarity.

The rules for $\leftrightarrow$ and $\otimes$ are polarity aware [3] in the sense that they aim to avoid a transformation that introduces unnecessary tautologies.

Example 1. Consider the formula $a \leftrightarrow (b \leftrightarrow c)$ and its transformation using the above rules:

$$\begin{array}{ll}
\langle \{ \{ (a \leftrightarrow (b \leftrightarrow c))^{\mathsf{t}} \}_\epsilon \}, (a \leftrightarrow (b \leftrightarrow c)) \rangle & \Rightarrow \\
\langle \{ \{ a^{\mathsf{f}}, (b \leftrightarrow c)^{\mathsf{t}} \}_\epsilon, \{ a^{\mathsf{t}}, (b \leftrightarrow c)^{\mathsf{f}} \}_\epsilon, (b \leftrightarrow c) \rangle & \Rightarrow \\
\langle \{ \{ a^{\mathsf{f}}, b^{\mathsf{t}}, c^{\mathsf{f}} \}_\epsilon, \{ a^{\mathsf{f}}, b^{\mathsf{f}}, c^{\mathsf{t}} \}_\epsilon, \{ a^{\mathsf{t}}, b^{\mathsf{f}}, c^{\mathsf{f}} \}_\epsilon, \{ a^{\mathsf{t}}, b^{\mathsf{t}}, c^{\mathsf{t}} \}_\epsilon, \epsilon \rangle &
\end{array}$$

leading to the clauses (cf. Clause Generation below):

$$\{\neg a, b, \neg c\}, \{\neg a, \neg b, c\}, \{a, \neg b, \neg c\}, \{a, b, c\}.$$

Here $\neg(b \leftrightarrow c)$ is translated as $(\neg b \vee \neg c) \wedge (b \vee c)$ rather than $(\neg b \wedge c) \vee (b \wedge c)$ as in the standard transformation. If the standard transformation had been applied then distributivity would have created the tautological clauses $\neg b \vee b$ and $c \vee \neg c$.

Also the rules for quantification are dual to each other and so the rule for one quantifier always also contains the one for the other just with the opposite polarity. In this paragraph, we only describe the "natural", positive case of each rule. The rule for universal quantification simply drops this quantification as variables are assumed universally quantified in the resulting clauses. The rule for existential quantification is the only rule that extends the substitution. We use the following example to explain the additional condition on this extension.

Example 2. Consider the formula $\varphi = \exists x.\forall y.\exists z.p(x, y, z)$ and its transformation using the above rules:

$$\begin{array}{ll}
\langle \{ \{ (\exists x.\forall y.\exists z.p(x,y,z))^{\mathsf{t}} \}_\epsilon \}, \exists x.\forall y.\exists z.p(x,y,z) \rangle & \Rightarrow \\
\langle \{ \{ (\forall y.\exists z.p(x,y,z))^{\mathsf{t}} \}_{\{x \mapsto a\}} \}, \forall y.\exists z.p(x,y,z) \rangle & \Rightarrow \\
\langle \{ \{ (\exists z.p(x,y,z))^{\mathsf{t}} \}_{\{x \mapsto a\}} \}, \exists z.p(x,y,z) \rangle & \Rightarrow \\
\langle \{ \{ p(x,y,z)^{\mathsf{t}} \}_{\{x \mapsto a, z \mapsto f(y)\}} \}, p(x,y,z) \rangle & \Rightarrow \\
\langle \{ \{ p(x,y,z)^{\mathsf{t}} \}_{\{x \mapsto a, z \mapsto f(y)\}} \}, \epsilon \rangle & \Rightarrow
\end{array}$$

The final product is the unit clause $p(a, y, f(y))$.

In this example, we introduce a Skolem function f for z taking the single variable y even though the free variables of $\exists z.p(x, y, z)$ include x also. This is because $\{(\exists z.p(x, y, z))\}_{\{x \mapsto a\}}$ stands for $\exists z.p(a, y, z)$ as the substitution is only delayed. Therefore, the rule for existential quantification must take the substitution into consideration.

Clause Generation. Given a terminal configuration $\langle \{D^i_{\sigma_i}\}, \epsilon\rangle$ the set of clauses generated is given as follows

$$\{(\bigvee \mathsf{form}(\varphi^\star))\sigma_i \mid \varphi^\star \in D^i_{\sigma_i}\}$$

i.e. the sign and substitution are applied to each (necessarily atomic) formula in the sequent.

Simplification. Whenever a sequent D_σ is constructed, simple tautologies and redundant formulas are eliminated. It means that

1. if D contains multiple occurrences of a signed formula, only one occurrence is kept in D;
2. if D contains $\top^{\mathsf{t}}$ or $\bot^{\mathsf{f}}$, D_σ is not added to Γ;
3. if D contains a signed formula $\bot^{\mathsf{t}}$ or $\top^{\mathsf{f}}$, this signed formula is removed from D.

Table 2: The reduction (difference) in the number of Skolem symbols introduced by VCNF compared to clausification previously implemented in Vampire.

Reduction	1	2	3	4	16	314
Number of problems	111	125	141	42	2	1

These rules are not required for replacing sequents, however they simplify formulas and make the resulting set of clauses smaller. A further simplification of removing *tautological sequents* is discussed in Section 6.

Correctness. It should be clear that the rules are terminating as Δ is bounded by the size of φ. It should also be clear that on termination Γ contains sequents consisting of sign atoms only, as the rules will eventually deconstruct all the complex (sub)formulas. Furthermore, whenever subformula φ' is handled by a rule it will no longer be presented in Γ, making the process linear in the size of the original formula φ. Finally, the transformation of φ leads to an equisatisfiable clausal form.

Lemma 1. Given $\langle \Gamma, \Delta \rangle \Rightarrow \langle \Gamma', \Delta' \rangle$ and a interpretation $\mathcal{I}$

1. if $\mathcal{I} \models \Gamma'$ then $\mathcal{I} \models \Gamma$
2. If $\mathcal{I} \models \Gamma$ then there exists $\mathcal{I}'$ that extends $\mathcal{I}$ on fresh symbols such that $\mathcal{I}' \models \Gamma'$

Theorem 1. For any formula φ in ENNF and finite set of sequents Γ if $\langle \{\{\varphi^{\mathsf{t}}\}_\epsilon\}, \varphi \rangle \Rightarrow^* \langle \Gamma, \epsilon \rangle$ then Γ is satisfiable if and only if φ is.

3.2 A note on reusing Skolem functions

One important additional remark is in order on how to interpret the quantifier rules of Figure 1. As with the other rules, there may be in general more than one sequent with an occurrence of the subformula $\varphi^\star$. However, to keep the signature from growing excessively a *single* new Skolem function can and should be introduced jointly for all those occurrences which share the same expression $\varphi\sigma$, i.e. those occurrences which agree on how the free variables of φ are getting bound. Note that it is not necessarily the case that the substitutions are the same in each sequent as they may contain additional variables.

The following is the conjecture formula of the `SYN723+1.p` problem from the TPTP library [6] which demonstrates the advantage of reusing Skolem functions as facilitated by VCNF.

$$\begin{aligned}(\exists x.\forall y.(p(x) \leftrightarrow p(y)) \leftrightarrow ((\exists x.q(x) \leftrightarrow \forall y.r(y)) \leftrightarrow ((\exists x.\forall y.(q(x) \leftrightarrow q(y))\\ \leftrightarrow (\exists x.r(x) \leftrightarrow \forall y.s(y))) \leftrightarrow (\exists x.\forall y.(r(x) \leftrightarrow r(y))\\ \leftrightarrow ((\exists x.s(x) \leftrightarrow \forall y.p(y)) \leftrightarrow (\exists x.\forall y.(s(x) \leftrightarrow s(y)) \leftrightarrow (\exists x.p(x) \leftrightarrow \forall y.q(y))))))))\end{aligned}$$

Because of the way quantification is interleaved with equivalences, the various subformulas of this formulas need to get copied and Skolemised separately by the usual transformation techniques. VCNF, on the other hand, Skolemises each subformula only once for each set of bindings for the subformulas' free variables. On this particular example, we obtained 330 Skolem functions for our old clausification algorithm compared to mere 16 with VCNF, i.e. more than a 20-fold reduction.

To estimate how often Skolem function reuse can be useful in practice we ran VCNF and the previous clausification algorithm of Vampire on the 9128 problems from the TPTP library (version 6.3.0) which are in first-order form. On 422 problems of these VCNF was able to produce fewer Skolem functions. Table 2 provides a "histogram" perspective on these problems, grouping them by the extent of the reduction. We remark that in the current setting, a necessary condition for a reduction to occur is the alternation of quantifiers and (non-)equivalences in the problem. Formula sharing (see Section 6) which we plan for future implementation has the potential to lead to further reduction possibilities.

4 Naming Subformulas

We now show how the previous algorithm can be used to name subformulas. The general idea is the following. Given a finite set of sequents Γ, if the number of instances of a signed formula $\varphi^\star$ reaches a certain threshold then φ is named.

4.1 Updating the Algorithm

We begin by updating the notion of configuration to include a subformula count C which maps subformulas to the number of times they occur *at a top level* in Γ. The rules in Figure 1 can then be updated to update this counter. For example, the rule for $\leftrightarrow$ becomes

$$\begin{gathered}\langle\{\{D^i,(\varphi_1\leftrightarrow\varphi_2)^{\mathsf{t}}\}_{\sigma_i},\{D^j,(\varphi_1\leftrightarrow\varphi_2)^{\mathsf{f}}\}_{\sigma_j},D^k_{\sigma_k}\},(\varphi_1\leftrightarrow\varphi_2).\Delta,\{(\varphi_1\leftrightarrow\varphi_2)\mapsto k\}\cup C\rangle\\ \Rightarrow\\ \langle\{\{D^i,\varphi_1^{\mathsf{f}},\varphi_2^{\mathsf{t}}\}_{\sigma_i},\{D^i,\varphi_1^{\mathsf{t}},\varphi_2^{\mathsf{f}}\}_{\sigma_i},\{D^j,\varphi_1^{\mathsf{f}},\varphi_2^{\mathsf{f}}\}_{\sigma_j},\{D^j,\varphi_1^{\mathsf{t}},\varphi_2^{\mathsf{t}}\}_{\sigma_j},D^k_{\sigma_k}\},\Delta.\varphi_1.\varphi_2,\\ \{\varphi_1\mapsto 2k,\varphi_2\mapsto 2k\}\cup C'\rangle\end{gathered}$$

where C' updates the formula count for formula occurring in D^i and D^j. Note that we drop the count for $(\varphi_1 \leftrightarrow \varphi_2)$ as this formula is being removed from Γ. Additionally, in our current setting without formula sharing (see Section 6) we are not keeping track of the occurrences and their count of the subformulas φ_1 and φ_2 until they are unwrapped.

Then a new rule is introduced that performs naming as follows.

$$\begin{gathered}\langle\{\{D^i,\varphi[\mathbf{y}]^{\mathsf{t}}\}_{\sigma_i},\{D^j,\varphi[\mathbf{y}]^{\mathsf{f}}\}_{\sigma_j},D^k_{\sigma_k}\},\Delta,\{\varphi[\mathbf{y}]\mapsto n\}\cup C\rangle\\ \textit{if } n>\textit{threshold}\ \Rightarrow\\ \langle\{\{D^i,P_\varphi(\mathbf{y})^{\mathsf{t}}\}_{\sigma_i},\{D^j,P_\varphi(\mathbf{y})^{\mathsf{f}}\}_{\sigma_j},D^k_{\sigma_k},\{P_\varphi(\mathbf{y})^{\mathsf{t}},\varphi^{\mathsf{f}}\}_\epsilon,\{P_\varphi(\mathbf{y})^{\mathsf{f}},\varphi_{\mathsf{t}}\}_\epsilon\},\Delta,C\rangle\end{gathered}$$

If φ only appears as φ^{t} then only the sequent $\{P_\varphi(\mathbf{y})^{\mathsf{t}},\varphi^{\mathsf{f}}\}_\epsilon$ is added, similarly for φ^{f} and $\{P_\varphi(\mathbf{y})^{\mathsf{f}},\varphi_{\mathsf{t}}\}_\epsilon$. This is the polarity-aware naming discussed earlier.

Finally, the rules are now non-deterministic as one could be in a situation where the naming rule and some other rule is applicable. In this case the naming rule should always be preferred, i.e. we name as soon as we are allowed to by the threshold.

5 Experiments

Comparing two different algorithms in first-order theorem proving is very hard. The reason is that the performance of a theorem prover depends on many options, which may affect the algorithms in different ways. It may turn out that one of the algorithms is better for some strategies (that is, combinations of parameter values), while the other one is better for other strategies. Thus, fixing a small number of strategies may result in a biased conclusion. To avoid this potential bias we used the methodology based on varying strategies. Further, we only focused on problems hard for Vampire.

We took a subset of 2,307 problems from the TPTP library [6] previously established hard for Vampire. We randomly generated strategies by flipping values of various options that define how the prover searches for a refutation. Each strategy was cloned into two, one using

VCNF and the other the previous clausification algorithm. Such a pair of strategies was run on a randomly selected hard problem. In total, we ran 130 000 pairs.

In total, a strategy using VCNF succeeded 6802 times and a strategy using the previous clausification 6514 times (which confirms that the problems were hard, since only about 5% of all runs were successful). There were 812 cases where only the VCNF variation succeeded on a problem compared to 524 cases where only the previous clausification led to a solution. This demonstrates that VCNF is a viable alternative to the standard clausification and, indeed, tends to help Vampire to find more solutions than the previous approach.

6 Tautology Detection and Elimination

One advantage of the VCNF algorithm is the ability to detect and discard tautological sequents on the fly.

Tautologies. A sequent D_σ is a *tautology* if it contains both φ^{t} and φ^{f}, i.e. if it is of the form $\{\varphi^{\mathsf{t}}, \varphi^{\mathsf{f}}, \ldots\}$. Such sequents can be discarded as soon as they are observed.

A tautological sequent will ultimately give rise to a set of tautological clauses. A clause originating from the clausification of $\{\varphi^{\mathsf{t}}, \varphi^{\mathsf{f}}, \ldots\}$ has the form $C \vee D \vee \ldots$ where $C \in \mathit{CNF}(\varphi)$ and $D \in \mathit{CNF}(\neg\varphi)$ thus $\varphi \rightarrow C$ and $\neg\varphi \rightarrow D$ and $C \vee D$ must be a tautological clause.

We note that such tautologies may not appear in the final clause set if subformula naming is used at any point during the process. Furthermore, the introduced definitions would not be tautologies and would need to be identified and removed using techniques such as *pure predicate elimination*[1].

Therefore, there are two main advantages to this removal of tautological sequents:

1. we avoid additional processing effort where they could otherwise be removed at the end,
2. we detect tautologies that may otherwise not be detected at the end if naming is used.

A final advantage is that removing tautologies early on makes the (effective) subformula occurrences counts (Section 4) more precise and therefore gives the naming mechanism more information to base its (however heuristical in nature) naming decisions on.

Source of Tautologies. The next question is where tautological sequents come from? There are (in our current setting) two possible sources:

1. a naive transformation of (non)equivalences,
2. they may occur in the input from the outset.

The first case is avoided via the polarity-aware transformation discussed earlier (Section 3). Therefore, only the second case is applicable here. In the case where φ is a literal, Vampire will detect the tautology as literals are perfectly shared in Vampire, i.e. their syntactic equivalence can be checked by pointer equivalence. As discussed below, such cases of tautologies involving literals do occur in practice.

Formula Sharing. In order to effectively detect the second case in general, an extra implementation trick is needed. In particular, we need to be able to recognise (in constant time) that the individual occurrences of φ indeed refer to the same formula. This does not seem to be possible with the standard (tree-like) representation of formulas. Therefore, a shared (dag-like) formula representation is required.

Vampire does not currently implement formula sharing (but this is planned for the future). There are certain technical issues that must be overcome. Firstly, formula sharing should be

[1]This is the process of detecting predicates that are only used with a single polarity and removing any clause containing them as they can be trivially satisfied.

modulo associativity and commutativity of $\wedge$, $\vee$, and $\otimes$ and commutativity of $\leftrightarrow$. But this is non-trivial, consider the formulas

$$\varphi_1 = p(x,y) \vee q(y) \vee s(x) \quad \textit{and} \quad \varphi_2 = p(x,y) \vee q(y) \quad \textit{and} \quad \varphi_3 = q(y) \vee s(x)$$

should φ_1 be represented as $\varphi_2 \vee s(x)$ or $p(x,y) \vee \varphi_3$? In either case the relationship between φ_1 and one of the other formulas will be lost.

Secondly, in the presence of first-order variables the possibility to share formulas which differ only by variable renaming becomes appealing. Moreover, since VCNF is naturally capable of processing a formula occurrence in the context of a general substitution σ, a possibility arises for an even more compact sharing structure aware of the "instance of" relation between formulas. Overcoming the technical challenges lying behind efficient implementation of these ideas is subject to future work.

As a further point, we note that the addition of formula sharing will change how the previous rules (Figure 1) operate as the list of subformulas to expand can no longer be simply maintained by pushing newly observed subformulas to the end of Δ. Instead, a more general order specified by the condition that it *preserves the subformula relation* will need to be employed.

Immediate tautology removal in practice. The TPTP library [6] contains a problem `SYN007+1.014.p` which can easily be solved by Vampire with VCNF but not with its previous clausification algorithm. This is Problem 71 from a collection by Pelletier [4] and consists of a single conjecture formed by a chain of equivalences:

$$p_1 \leftrightarrow (p_2 \leftrightarrow \ldots \leftrightarrow (p_{14} \leftrightarrow (p_1 \leftrightarrow (p_2 \leftrightarrow \ldots \leftrightarrow p_{14})))).$$

The key to success of VCNF on this problem lies with immediate tautology removal. That is because the clausal form of this problem consists of 2^{28} clauses (a huge number) when counting tautologies while the number is "only" 2^{14} (still manageable) if tautologies are discarded. Note that the property of eliminating *immediately* (before the expansion is completed) is important here, since 2^{28} clauses is too many to even just generate and discard on the fly.

7 Context Inlining

Sequents carry a substitution which defines a *context* for the free variables occurring in formulas of the sequent. This section considers whether this context can be *inlined* into the sequent to positive effect. There are two kinds of inling we consider:

1. Propagating (part of) this context to a subformula definition when formula occurrences being named share the relevant part of the context.
2. The possibility to reuse Skolem functions already introduced for the sake of a different occurrence when the relevant part of the context of the current occurrence is an instance (a special case) of the other occurrence's context.

The first case aims to reduce the arity of the new symbol introduced during naming and the second case avoids the introduction of an additional Skolem function. We note that these extensions are not currently implemented but illustrate further benefits of the top-down approach.

7.1 Inlining for Naming

The overall idea is to move as much to the definition during naming as possible to reduce the arity of the fresh predicate symbol. Let us begin with an illustrative example, consider the configuration $\langle\{\{D^i, \varphi[x_1,\ldots,x_n,y]^\star\}_{\sigma_i}\}, \Delta, C\rangle$ where we decide to name φ. Furthermore, assume that $x_1,\ldots,x_n \notin \mathsf{dom}(\sigma_i)$ and $y \mapsto t \in \sigma_i$. The variable y is the relevant part of the context that we will inline. Now instead of naming φ we name $\varphi\{y \mapsto t\}$, i.e. the instance of φ in which the binding to y has been inlined. This reduces the arity of the name if and only if the free variables of t are amongst $x_1, \dots, x_n$ (one such special case is when t is a constant).

In general, more than one variable can be shared by all the contexts of the occurrences of φ. We may wish to inline such a subset of these that the arity of the introduced name would be minimal. This is a non-trivial optimisation problem. Greedy solutions might not work and it may be necessary to go against the gradient to achieve optimum. Consider the following formula

$$\forall y_1, y_2. \exists x_1, \dots x_n . \varphi(x_1, \dots, x_n)$$

here when naming φ if we inline (the skolem term for) x_1 we will increase the arity of the name (since the skolem term depends on y_1 and y_2), but inlining all of x_i will finally reduce the arity to 2. This shows that the decision of whether to inline a particular variable is not independent.

Finally, if not all occurrences of a formula share the same context to be inlined then the sets of occurrences could be split and named separately. The question is then whether having two names with few arguments is better than only one name with many.

Let us remark that a similar discussion concerning naming of formulas related by the "instance-of" relation appears in related work [1].

7.2 Inlining for Skolemization

Next we demonstrate how context for Skolem functions could be inlined using the formula $\varphi = a \leftrightarrow \exists x.(b \leftrightarrow \exists y.p(x,y))$, which could be processed by VCNF as follows:

$$\begin{aligned}
&\langle \{\{(a \leftrightarrow \exists x(b \leftrightarrow \exists y.p(x,y)))^{\mathsf{t}}\}_\epsilon\}, (a \leftrightarrow \exists x.(b \leftrightarrow \exists y.p(x,y)))\rangle && \Rightarrow\\
&\langle \{\{a^{\mathsf{t}}, (\exists x.(b \leftrightarrow \exists y.p(x,y)))^{\mathsf{f}}\}_\epsilon, \{a^{\mathsf{f}}, (\exists x.(b \leftrightarrow \exists y.p(x,y)))^{\mathsf{t}}\}_\epsilon\}, (\exists x(b \leftrightarrow \exists y \varphi))\rangle && \Rightarrow\\
&\langle \{\{a^{\mathsf{t}}, (b \leftrightarrow \exists y.p(x,y))^{\mathsf{f}}\}_\epsilon, \{a^{\mathsf{f}}, (b \leftrightarrow \exists y.p(x,y))^{\mathsf{t}}\}_{\{x \mapsto c\}}\}, (b \leftrightarrow \exists y.p(x,y))\rangle && \Rightarrow\\
&\langle \{\{a^{\mathsf{t}}, b^{\mathsf{f}}, (\exists y.p(x,y))^{\mathsf{f}}\}_\epsilon, \{a^{\mathsf{t}}, b^{\mathsf{t}}, (\exists y.p(x,y))^{\mathsf{t}}\}_\epsilon,\\
&\quad \{a^{\mathsf{f}}, b^{\mathsf{t}}, (\exists y.p(x,y))^{\mathsf{f}}\}_{\{x \mapsto c\}}, \{a^{\mathsf{f}}, b^{\mathsf{f}}, (\exists y.p(x,y))^{\mathsf{t}}\}_{\{x \mapsto c\}}\}, (\exists y.p(x,y))\rangle && \Rightarrow\\
&\langle \{\{a^{\mathsf{t}}, b^{\mathsf{f}}, p(x,y)^{\mathsf{f}}\}_\epsilon, \{a^{\mathsf{t}}, b^{\mathsf{t}}, p(x,y)^{\mathsf{t}}\}_{\{y \mapsto d(x)\}},\\
&\quad \{a^{\mathsf{f}}, b^{\mathsf{t}}, p(x,y)^{\mathsf{f}}\}_{\{x \mapsto c\}}, \{a^{\mathsf{f}}, b^{\mathsf{f}}, p(x,y)^{\mathsf{t}}\}_{\{x \mapsto c, y \mapsto d(c)\}}, \}, \epsilon\rangle && \Rightarrow
\end{aligned}$$

Context inlining for Skolem functions is demonstrated by the last step where we Skolemise the subformula $\exists y.p(x,y)$. Let us focus on two the positive occurrences for which we need to introduce a Skolem function. Since the first of these occurrences happens in the context where x is unbound, we introduce a term with a dependency on this variable, adding $y \mapsto d(x)$. For the second occurrence, the context contains the binding $x \mapsto c$ and thus, according to the last rule from Figure 1, y should be bound to a new Skolem constant e. However, because this context is an instance of the first one, we can instead reuse the previously introduced Skolem term $d(x)$ and inline the binding $x \mapsto c$ into it. Thus we add the binding $y \mapsto d(c)$ and avoid the introduction of the constant e.

Note that when Skolemising a formula $\exists y.\varphi[y, \mathbf{x}]$, we could in principle always introduce the most general Skolem function $f(\mathbf{x})$, with a dependency on all the free variables of φ and inline the respective context of each occurrence to it. However, this would lead to introduction of Skolem functions with unnecessarily large arities. So similarly to inlining for naming the idea of inlining for Skolemisation leads to an optimisation problem (between the number of Skolem symbols introduces and their arities) which does not have an obvious optimum.

8 Conclusion

This paper has introduced a new top-down algorithm for clausal form generation for first-order formulas. The algorithm allows for the introduction of fewer Skolem functions as the context from equivalence expansion is preserved. The algorithm also captures an alternative approach to subformula naming that can reduce the size of the clausal form. Both of these results have been established experimentally. Two extensions to the algorithm are then considered. Firstly, we conjecture that formula sharing would enable further tautological sequent removal. Secondly, we describe two settings where the context built during VCNF could be inlined to reduce the arity of naming definitions or the number of Skolem functions required. Implementation of these extensions remains further work.

References

[1] N. Azmy and C. Weidenbach. Computing tiny clause normal forms. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pp. 109–125, 2013.

[2] L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *CAV 2013*, vol. 8044 of *Lecture Notes in Computer Science*, pp. 1–35, 2013.

[3] A. Nonnengart and C. Weidenbach. Computing small clause normal forms. In *Handbook of Automated Reasoning (in 2 volumes)*, pp. 335–367. Elsevier and MIT Press, 2001.

[4] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *J. Autom. Reasoning*, 2(2):191–216, 1986.

[5] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.

[6] G. Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.