nanoCoP: Natural Non-clausal Theorem Proving

Jens Otten
Department of Informatics, University of Oslo, Norway
jeotten@ifi.uio.no

Abstract

Most efficiently fully automated theorem provers implement proof search calculi that require the input formula to be in a clausal form, i.e. disjunctive or conjunctive normal form. The translation into clausal form introduces a significant overhead to the proof search and modifies the structure of the original formula. Translating a proof in clausal form back into a more readable non-clausal proof of the original formula is not straightforward. This paper presents a non-clausal automated theorem prover for classical first-order logic. It is based on a non-clausal connection calculus and implemented with a few lines of Prolog code. Working entirely on the original structure of the input formula yields not only a speed up of the proof search, but the resulting non-clausal proofs are also shorter.

1 Introduction

Automated Theorem Proving (ATP) in classical first-order logic is a core research area in the field of Automated Reasoning (see [Robinson and Voronkov, 2001] for an overview). It is concerned with the question whether a conjecture \( H \) is a logical consequence of a given set of axioms \( G_1, \ldots, G_n \), written \( G_1, \ldots, G_n \models H \), in which \( G_1, \ldots, G_n, H \) are first-order formulae. More specifically the objective in ATP is to find a proof for the validity of a given formula \( F \). A formula \( F \) is valid if and only if it evaluates to true for all possible interpretation of its function and predicate symbols.

According to the Deduction Theorem (for classical logic), \( G_1, \ldots, G_n \models H \) holds if and only if there is a proof for the validity of the formula \( (G_1 \land \ldots \land G_n) \Rightarrow H \). A formal description of the proof search algorithm is usually specified in form of a proof (search) calculus consisting of axioms and rules. Implementations of such proof calculi are called ATP systems or (automated theorem) provers.

Most efficiently fully automated theorem provers implement proof calculi that require the input formula to be in a clausal form, i.e. disjunctive or conjunctive normal form. Formulae that are not in clausal form are translated into clausal form in a preprocessing step. For example, the formula \( F \)

\[
P(a) \land (\forall y(P(y) \Rightarrow P(g(y)))) \lor (Q \Rightarrow Q) \lor (R \Rightarrow R) \Rightarrow P(g(a))
\]

has the disjunctive normal form (quantifiers are eliminated)

\[
\neg P(a) \lor (P(y) \land P(g(y))) \lor (Q \land R) \lor (P(y) \land \neg P(g(y))) \lor (Q \land \neg R) \\
\lor (P(y) \land \neg P(g(y))) \lor (Q \land R) \lor (P(y) \land \neg P(g(y))) \lor (Q \land \neg R) \\
\lor P(g(a)).
\]

While the use of a clausal form technically simplifies the proof calculi and their implementations, it has some fundamental disadvantages. The standard translation into clausal form as well as the definitional translation [Plaisted and Greenbaum, 1986; Eder, 1992], which introduces definitions for subformulae, cause a significant overhead for the proof search [Otten, 2010]. For example, the disjunctive normal form of \( F \), in which some parts of the formula are copied, has more than twice the size of the original formula. Furthermore, a translation into clausal form modifies the structure of the formula, hence, a translation of the clausal proof back into one of the original formula is not straightforward [Reis, 2015]. On the other hand, proof search with more “natural” non-clausal calculi, such as sequent or standard tableau calculi [Gentzen, 1935; Hähnle, 2001] is less efficient.

This paper describes the non-clausal prover nanoCoP for classical first-order logic [Otten, 2016]. By performing the proof search on the original structure of the input formula, it combines the advantages of more natural non-clausal calculi with the efficiency of a goal-oriented connection-based proof search. The prover is based on a non-clausal connection calculus [Otten, 2011] (Section 2) that generalizes the clausal connection calculus [Bibel, 1983; 1987] and is implemented in a very compact way (Section 3). It follows the lean methodology already used for the clausal connection prover leanCoP, whose minimal Prolog source code is shown in Figure 1. An experimental Evaluation (Section 4) indicates a solid performance of nanoCoP.

```
prove([1, S], ) :- 
+ member(scut, S) -> prove([[-#]], [], S) ;
lit(#, C, _) -> prove([C, [-#]], [], S).
prove([1, S], ) :- member(comp(L), S), I=L -> prove([L], []). 
(member(comp(_), S); retract(p)) -> J is I+1, prove(J, S).
prove([L|C], F, I, Q, S) :- 
+ (member(A, [L|C]), member(B, P), A=N, L=N) -> (member(D, Q), L=N) ;
member(E, P), unify_with_occurs_check(E, N) ;
lit(N, F, N), length(P, K), K<I, true ;
lit(#, C, _), lit(#, B, _), (+p -> assert(p, fail)), prove([L|F], Q, S, ) ;
(member(comp(_), S); retract(p)) -> J is I+1, prove(J, S).
```

Figure 1: Source code of the leanCoP 2.0 core prover
2 The Non-clausal Connection Calculus

The standard notation for first-order formulae is used. Terms (denoted by $t$) are built up from functions ($f, g, h, i$), constants ($a, b, c$), and variables ($x, y, z$). An atomic formula ($A$) is built up from predicate symbols ($P, Q, R, S$) and terms; a (first-order) formula ($F, G, H$) is built up from atomic formulae, the connectives $\land, \lor, \Rightarrow$, and the first-order quantifiers $\forall, \exists, \forall, \forall$. A literal $L$ has the form $A$ or $\neg A$. Its complement $\overline{L}$ is $A$ if $L$ is of the form $\neg A$; otherwise $\overline{L}$ is $\neg L$.

A term substitution $\sigma$ assigns terms to variables. A formula in clausal form has the form $\exists x_1 \ldots \exists x_n (C_1 \lor \ldots \lor C_m)$, where each clause $C_i$ is a conjunction of literals $L_1, \ldots, L_m$. It is represented as a set of clauses $\{C_1, \ldots, C_n\}$, called a (clausal) matrix. A polarity $0$ or $1$ is used to represent negation, i.e. literals of the form $\neg A$ and $A$ are represented by $A^0$ and $A^1$, respectively. A connection is a set $\{A^0, A^1\}$ of literals with the same predicate symbol but different polarities.

In a non-clausal matrix, a clause consists of literals and (sub)matrices. Let $F$ be a formula and $pol$ be a polarity. The non-clausal matrix $M(F^{pol})$ of a formula $F$ is a set of clauses, in which a clause is a set of literals and matrices, and is defined according to Table 1. In Table 1, $x^*$ is a new variable, $t^*$ is the skolem term $f^*(x_1, \ldots, x_n)$ in which $f^*$ is a new function symbol and $x_1, \ldots, x_n$ are the free variables in $\forall x G$ or $\exists x G$. In $G[x \backslash t]$ all free occurrences of $x$ in $G$ are replaced by $t$. The non-clausal matrix $M(F)$ of a formula $F$ is the matrix $M(F^0)$. In the graphical representation its clauses are arranged horizontally, while the literals and matrices of each clause are arranged vertically. For example, the formula $F_{#y}$ of Section 1 has the simplified (i.e. redundant brackets are removed) non-clausal matrix $M_{#y} = M(F_{#y})$:

$$\{\{P(a)^1\}, \{\{P(y)^0, P(g(y))^1\}\}, \{\{Q^0\}, \{Q^1\}\}, \{\{R^0\}, \{R^1\}\}, \{P(g(g(a))^0)\}\}.$$ 

The graphical representation of the matrix $M_{#y}$ is depicted in Figure 2. It already contains a clause copy with the fresh variable $y$ and represents a non-clausal connection proof using the term substitution $\sigma$ with $\sigma(y) = a$ and $\sigma(y^*) = g(a)$; literals of each connection are connected with a line.

Compared to the formal clausal connection calculus (Otten and Bibel, 2003), a decomposition rule is added to the non-clausal calculus and the extension rule is slightly modified.

<table>
<thead>
<tr>
<th>Type</th>
<th>$F^{pol}$</th>
<th>$M(F^{pol})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$A^{pol}$</td>
<td>${A^{pol}}$</td>
</tr>
<tr>
<td></td>
<td>$(\neg G)^{pol}$</td>
<td>$M(G^{1-pol})$</td>
</tr>
<tr>
<td></td>
<td>$(G \land H)^0$</td>
<td>${{M(G)^0}, {M(H)^0}}$</td>
</tr>
<tr>
<td></td>
<td>$(G \lor H)^0$</td>
<td>${{M(G)^0}, {M(H)^0}}$</td>
</tr>
<tr>
<td>$\beta$</td>
<td>$(G \Rightarrow H)^0$</td>
<td>${{M(G)^0}, {M(H)^0}}$</td>
</tr>
<tr>
<td></td>
<td>$(G \Rightarrow H)^0$</td>
<td>${{M(G)^0}, {M(H)^0}}$</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>$(\forall x G)^1$</td>
<td>$M(G[x \backslash t^*])$</td>
</tr>
<tr>
<td></td>
<td>$(\exists x G)^0$</td>
<td>$M(G[x \backslash t^*])$</td>
</tr>
<tr>
<td>$\delta$</td>
<td></td>
<td>$M(G[x \backslash t^*])$</td>
</tr>
</tbody>
</table>

Table 1: The definition of the non-clausal matrix

Figure 2: Graphical representation of a non-clausal matrix and its non-clausal connection proof

A clause $C$ contains a literal $L$ if and only if (iff) $L \in C$ or $C'$ contains $L$ for $M' \subseteq C' \subseteq M$. A clause $C$ is $\alpha$-related to a literal $L$ iff it contains a matrix $M'$ with $\{C', C''\} \subseteq M'$ for clauses $C', C''$, such that $C'$ contains $L$ and $C''$ contains $C$ (or $C'' \subseteq C'$). A copy of the clause $C$ in the matrix $M$ is made by renaming all free variables in $C$. $M[C_1 \setminus C_2]$ denotes the matrix $M$, in which the clause $C_1$ is replaced by $C_2$. $C'$ is a parent clause of $C$ iff $M' \subseteq C'$ and $C' \subseteq M'$ for some $M'$.

Let $M$ be a matrix and $Path$ be a set of literals. $C$ is an extension clause (e-clause) of the matrix $M$ with respect to a set of literals Path iff either (a) $C$ contains a literal of Path, or (b) $C$ is $\alpha$-related to all literals of Path occurring in $M$ and $C$ has a parent clause, it contains a literal of Path. In the $\beta$-clause of $C_2$ with respect to $L_2$, denoted by $\beta$-clause($L_2$), $L_2$ and clauses that are $\alpha$-related to $L_2$ are deleted from $C_2$.

The non-clausal connection calculus (Otten, 2011), which is sound and complete, is shown in Figure 3. Therein, $M$ is a non-clausal matrix, $C$ is a (subgoal) clause or $\varepsilon$ (a special empty symbol) and (the active) Path is a set of literals or $\varepsilon$; $\sigma$ is a term substitution. A non-clausal connection proof of $M$ is a non-clausal connection proof of $\varepsilon, M, \varepsilon$.

The analytic, i.e. bottom-up, proof search is carried out in the same way as in the clausal calculus. Additional backtracking might be required when selecting $C_1$ in the decomposition rule, but no backtracking is required when selecting $M_1$. The rigid, i.e. single, term substitution $\sigma$ is calculated whenever a connection is identified in a reduction or extension rule. On formulae in clausal form, the non-clausal connection calculus coincides with the clausal connection calculus.

Figure 3: The non-clausal connection calculus
3 The nanoCoP Theorem Prover

The implementation of the non-clausal connection calculus of Figure 3 follows the lean methodology [Beckert and Posegga, 1995], which is already used for the clausal connection prover leanCoP [Otten and Bibel, 2003]. It uses very compact Prolog code to implement the basic calculus and adds a few essential optimization techniques in order to prune the search space. The resulting natural nonclausal connection prover nanoCoP is available under the GNU General Public License and can be downloaded at http://www.leancop.de/nanocop/.

In a first step, the input formula \( F \) is translated into a non-clausal (indexed) matrix \( M(F) \) according to Table 1; redundant brackets are removed [Otten, 2011]. Additionally, every \((sub-)clause (I,V):C and (sub-)matrix J: M \) is marked with a unique index \( I \) or \( J \); clause \( C \) is also marked with a set of variables \( V \) that are newly introduced in \( C \) but not in any subclause of \( C \). Atomic formulae are represented by Prolog terms, term variables by Prolog variables and the polarity 1 by "-". Sets, e.g. clauses and matrices, are represented by Prolog lists (representing multisets). For example, the matrix \( M_\beta \) from Section 2 is represented by the Prolog term

\[
\begin{align*}
(0^K)^\{[]:[-(p(a))], \\
(1^K)^\{[2^K]:[(3^K)^\{[]:[y]:[p(y),-(p(g(y)))]]], \\
4^K:[(5^K)^\{[]:[q],(6^K)^\{[]:[-q]], \\
7^K:[(8^K)^\{[]:[r],(9^K)^\{[]:[-r]], \\
(10^K)^\{[]:[p(g(g(a)))]])
\end{align*}
\]

in which the Prolog variable \( K \) is used to enumerate clause copies. In the second step, the matrix \( M = M(F) \) is written into Prolog’s database. For every literal \( Lit \) in \( M \) the fact

\[
lit(Lit,Cla,C,G)
\]

is asserted into the database where \( Cla \in M \) is the (largest) clause in which \( Lit \) occurs and \( Cla \) is the \( \beta \)-clause of \( Cla \) with respect to \( Lit \). \( G \) is set to \( g \) if the smallest clause in which \( Lit \) occurs is ground, i.e. does not contain any variables; otherwise \( G \) is set to \( n \). Storing literals of \( M \) in the database in this way is called lean Prolog technology [Otten, 2010] and integrates the advantages of the Prolog technology approach [Stickel, 1988] into the lean theorem proving framework. No other modifications or simplifications of the original formula are done during these two preprocessing steps.

The nanoCoP source code is shown in Figure 4. As an optimization, it uses positive start clauses [Otten, 2011], calculated by the predicate positiveC(Cla1).

The predicate prove(Mat,PathLim,Set,Proof) implements the start rule. Mat is the matrix generated in the preprocessing step. PathLim is the maximum size of the active path used for iterative deepening, and Proof contains the returned connection proof. Set is a list of options used to control the restricted backtracking technique [Otten, 2010] and may contain "cut" and "comp (I)" for \( I \in \mathbb{N} \). In order to achieve completeness, nanoCoP performs an iterative deepening on the size of the active path implemented within the second Prolog clause of this prove predicate.

The predicate prove(Cla,Mat,Path,PathI,PathLim,LEM,Set,Proof) implements the axiom, the decomposition rule, the reduction rule, and the extension rule of the non-clausal connection calculus of Figure 3. Cla, Mat, and Path represent the subgoal clause \( C \), the (indexed) matrix \( M \) and the (active) Path. The term substitution \( \sigma \) is stored implicitly by Prolog. The indexed path \( Path \) contains the indices of all clauses and matrices that contain literals of \( Path \); it is used for calculating extension clauses. The list \( LEM \) is used for the lemmata rule and contains all literals that have already been "solved" [Otten, 2010]. This prove predicate succeeds iff there is a connection proof for the tuple \( Cla, Mat, Path \) with \( Path < PathLim \). In this case Proof returns a connection proof. The input matrix Mat has to be stored in Prolog’s database as explained above.

Finally, the last predicate prove_ec(ClaB,MatI,Path,PathI,ClaB1,SetI) is used to calculate extension clauses. Starting with the (largest possible) extension clause \( Cla \), its \( \beta \)-clause \( ClaB \), the current (indexed) matrix Mat, and the indexed path \( PathI \), it returns an appropriate extension clause \( Cla \), copies it into \( Mat \) and returns its \( \beta \)-clause \( ClaB1 \) and the new (indexed) matrix \( MatI \).

nanoCoP uses additional optimization techniques that are already used in the clausal connection prover leanCoP: regularity, lemmata, and restricted backtracking [Otten, 2010].

Figure 4: The source code of the nanoCoP prover.
4 Experimental Evaluation

The following evaluations were conducted on a 3.4 GHz Xeon system with 4 GB of RAM running Linux 3.13.0 and ECLiPSe Prolog 5.10. The CPU time limit for all proof attempts was set 100 seconds.

The following formula $F_i$ is a slightly extended version of the formula $F_{\#}$ of Section 1, where $f^n, g^n, h^n$, and $i^n$ are abbreviations for $n$ nested applications of these functions:

$$F_n \equiv P(a) \land (\neg((Q(f^n(c)) \land \forall x(Q(f(x))) \Rightarrow Q(c)) \lor \neg((R^n(c)) \land \forall x(R(h(x))) \Rightarrow R(c)) \lor \neg((S^n(c)) \land \forall x(S(i(x))) \Rightarrow S(c)) \lor \forall y(P(y) \Rightarrow P(g(y)))) \Rightarrow \exists z P(g^n(z)).$$

Table 2 shows the timings on this (valid) formula class for $n=10, n=30,$ and $n=90$ for the following provers: the lean (non-clausal) tableau prover leanTAP [Beckert and Posegga, 1995], the resolution prover Prover9 [McCune, 2005], the superposition prover E [Schulz, 2002] (using options “-auto -ttptp3-format”), the clausal connection prover leanCoP [Otten and Bibel, 2003; Otten, 2010], and nanoCoP. The leanCoP core prover with the standard (“[nodef]”) and the definitional (“[def]”) translation into clausal form were tested. For nanoCoP restricted backtracking was switched off (Set=[1]). Times are given in seconds; the size of the returned proof tree, i.e. the number of nodes of the proof tree, is given in brackets.

The last line of Table 2 shows the number of proved problems of all 5051 first-order (FOF) problems in the TPTP library v3.7.0 [Sutcliffe, 2009]. For leanTAP, leanCoP, and nanoCoP, the required equality axioms were added in a preprocessing step (which is included in the timings). The nanoCoP core prover performs significantly better than both clausal form translations of the leanCoP core prover. Furthermore, 40%/51% of the proofs found by nanoCoP are shorter than those of leanCoP [nodef]/[def], respectively; as many of the problems in the TPTP library are “mostly” in a clausal form, 56%/47% of the proofs have the same size. The proofs of nanoCoP are up to 96%/74% shorter than those of the leanCoP versions [nodef]/[def], respectively.

The “full” leanCoP 2.2 prover, which additionally uses a strategy scheduling [Otten, 2010], proves 1710 TPTP problems. The version of nanoCoP using a restricted backtracking strategy, i.e. Set=[cut, comp(6)], was tested as well; it proves 1485 TPTP problems.

Tests have also shown that the classical version of the non-clausal connection prover JProver [Schmitt et al., 2001], which is the only published non-clausal connection prover so far, has a lower performance than leanTAP.

Table 2: Results on formula class $F_n$ and the TPTP library

<table>
<thead>
<tr>
<th></th>
<th>leanTAP</th>
<th>Prover9</th>
<th>E</th>
<th></th>
<th>leanCoP 2.2</th>
<th>nanoCoP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.3</td>
<td>2009-02A</td>
<td>1.9</td>
<td></td>
<td>[nodef]</td>
<td>[def]</td>
</tr>
<tr>
<td>$F_{10}$</td>
<td>0.17 (128)</td>
<td>–</td>
<td>1.22 (2916)</td>
<td>–</td>
<td>–</td>
<td><strong>0.09 (45)</strong></td>
</tr>
<tr>
<td>$F_{30}$</td>
<td>–</td>
<td>–</td>
<td>84.57 (57628)</td>
<td>–</td>
<td>–</td>
<td><strong>0.12 (125)</strong></td>
</tr>
<tr>
<td>$F_{90}$</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td><strong>0.42 (365)</strong></td>
</tr>
<tr>
<td>TPTP</td>
<td>404</td>
<td>1611</td>
<td>2782</td>
<td>1134</td>
<td>1065</td>
<td><strong>1232</strong></td>
</tr>
</tbody>
</table>

5 Conclusion

Formal reasoning is a fundamental task in mathematics, computer science and many related fields. Since Frege published the first formal calculus for first-order logic in his *Begriffsschrift* [Frege, 1879], the development and implementation of efficient proof calculi has made significant progress. But up to now, most—if not all—efficient fully automated theorem provers for classical first-order logic translate the input formula into a clausal form. By using the standard translation the size of the resulting formula can grow exponentially with respect to the size of the original formula. Even a definitional translation increases the size of the formula, which results in a significant overhead for the proof search [Otten, 2010]. Both clausal form translations modify the structure of the formula, making it difficult to translate the (clausal) proof back into a proof of the original formula.

This paper introduced nanoCoP, the first efficient non-clausal connection proof for classical first-order logic. Using non-clausal matrices the proof search works directly on the original structure of the input formula and, thus, avoids a translation into any clausal form. This combines the advantages of more natural non-clausal sequent or tableau provers with the goal-oriented efficiency of connection provers.

Even though the non-clausal inferences introduce a slight overhead, nanoCoP outperforms both clausal form translations of the leanCoP core prover on a large set of TPTP problems. About half of the returned non-clausal proofs are up to 96% shorter than their clausal counterparts.

nanoCoP returns a compact non-clausal connection proof. A connection corresponds to a closed branch in the tableau calculus [Hähnle, 2001] or an axiom in the sequent calculus [Gentzen, 1935]. Hence, the translation into, e.g., a sequent proof is straightforward. The compact size of nanoCoP makes it also a suitable tool for the development of verifiably correct software [Otten and Bibel, 2017], as its correctness can be proven much more easily than that of an ATP system consisting of several thousand lines of source code.

Only few research work investigates non-clausal connection calculus and implementations. Other approaches [Andrews, 1981; Bibel, 1987; Hähnle et al., 2004; Kreitz and Otten, 1999] work (efficiently) only for ground formulae or their implementation is not available anymore [Issar, 1990].

Another important application of nanoCoP is its usage within non-classical logics, such as (first-order) intuitionistic or modal logic, for which the use of a clausal form is either not desirable or not possible. Hence, future work includes the combination of the non-clausal approach with the prefix (unification) technique for some non-classical logics, as already done for leanCoP [Otten, 2008; 2014]. In order to improve performance, further optimization techniques need to be integrated into nanoCoP, such as strategy scheduling [Otten, 2010], learning [Kaliszyk and Urban, 2015] or variable splitting [Antonsen and Waaler, 2007].

Acknowledgements

The author would like to thank Wolfgang Bibel for his helpful comments on a preliminary version of this paper.
References


