nanoCoP: A Non-clausal Connection Prover

Jens Otten

Institut für Informatik, University of Potsdam
August-Bebel-Str. 89, 14482 Potsdam-Babelsberg, Germany
jeotten@cs.uni-potsdam.de

Abstract. Most of the popular efficient proof search calculi work on formulae that are in clausal form, i.e. in disjunctive or conjunctive normal form. Hence, most state-of-the-art fully automated theorem provers require a translation of the input formula into clausal form in a preprocessing step. 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 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. By working entirely on the original structure of the input formula, the resulting non-clausal proofs are not only shorter, but can also be more easily translated into, e.g., sequent proofs. Furthermore, a non-clausal proof search is more suitable for some non-classical logics.

1 Introduction

Automated theorem proving in classical first-order logic is a core research area in the field of Automated Reasoning. Most efficient 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.[1] First-order formulae that are not in this form are translated into clausal form in a preprocessing step. While the use of a clausal form technically simplifies the proof search and the required data structures, it also has some disadvantages. The standard translation into clausal form as well as the definitional translation [19], which introduces definitions for subformulae, introduce a significant overhead for the proof search [14]. Furthermore, a translation into clausal form modifies the structure of the original formula and the translation of the clausal proof back into one of the original formula is not straightforward [20]. On the other hand, fully automated theorem provers that use non-clausal calculi, such as standard tableau or sequent calculi, are usually not suitable for an efficient proof search.

The present paper describes the non-clausal connection prover nanoCoP for classical first-order logic. By performing the proof search on the original structure of the input formula, it combines the advantages of more *natural* non-clausal provers with a more efficient *goal-oriented* connection-based proof search. The prover is based on a non-clausal connection calculus for classical first-order logic [15] (Sec. 2) that generalizes the clausal connection (tableau) calculus [4, 5]. This non-clausal calculus is implemented in a very compact way (Sec. 3) following the *lean* methodology. An experimental evaluation (Sec. 4) shows a solid performance of nanoCoP.

[1] In the core first-order category "FOF" at the most recent ATP competition, CASC-25, only the Muscadet prover implements a proof search that works on the original formula structure.

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 (denoted by A) is built up from predicate symbols (P,Q,R,S) and terms. A *(first-order) formula* (denoted by F,G,H) is built up from atomic formulae, the connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and the standard first-order quantifiers $\forall$ and $\exists$. 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 *connection* is a set $\{A,\neg A\}$ of literals with the same predicate symbol but different polarity. A *term substitution* σ assigns terms to variables. A formula in *clausal form* has the form $\exists x_1 \ldots \exists x_n (C_1 \vee \ldots \vee C_n)$, where each clause C_i is a conjunction of literals $L_1,\ldots,L_{m_i}$. It is usually represented as a set of clauses $\{C_1,\ldots,C_n\}$, which is called a (clausal) *matrix*. The *polarity* 0 or 1 is used to represent negation in a matrix, i.e. literals of the form A and $\neg A$ are represented by A^0 and A^1, respectively,

The non-clausal connection calculus uses non-clausal matrices. 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^{pol} is a set of clauses, in which a clause is a set of literals and (sub-)matrices, and is defined inductively 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 $\forall xG$ or $\exists xG$. 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 (sub-)matrices of each clause are arranged vertically. For example, the formula $F_\#$

$$P(a) \wedge (\neg((Q(f(f(c))) \wedge \forall x(Q(f(x)) \Rightarrow Q(x))) \Rightarrow Q(c)) \vee \forall y(P(y) \Rightarrow P(g(y)))) \Rightarrow \exists z P(g(g(z)))$$

has the simplified (i.e. redundant brackets are removed) non-clausal matrix $M_\# = M(F_\#)$:

$$\{\{P(a)^1\},\{\{\{Q(f(f(c)))^1\},\{Q(f(x))^0,Q(x)^1\},\{Q(c)^0\}\},\{\{P(y)^0,P(g(y))^1\}\}\},\{P(g(g(z)))^0\}\}\ .$$

The graphical representation of the matrix $M_\#$ is depicted in Figure 1. It already contains two clause copies using the fresh variables x' and y' and represents a non-clausal connection proof, in which the literals of each connection are connected with a line, using the substitution σ with $\sigma(x) = f(c)$, $\sigma(x') = c$, $\sigma(y) = \sigma(z) = a$, $\sigma(y') = g(a)$.

The axiom and the rules of the *non-clausal connection calculus* [15] are given in Fig. 2. It works on tuples "$C,M,Path$", where M is a non-clausal matrix, C is a (subgoal) clause or ε and (the active) $Path$ is a set of literals or ε; σ is a term substitution. A *non-clausal connection proof* of M is a non-clausal connection proof of $\varepsilon,M,\varepsilon$.

Table 1. The definition of the non-clausal matrix

type	F^{pol}	$M(F^{pol})$
atomic	A^0	$\{\{A^0\}\}$
	A^1	$\{\{A^1\}\}$
α	$(\neg G)^0$	$M(G^1)$
	$(\neg G)^1$	$M(G^0)$
	$(G \wedge H)^1$	$\{\{M(G^1)\},\{M(H^1)\}\}$
	$(G \vee H)^0$	$\{\{M(G^0)\},\{M(H^0)\}\}$
	$(G \Rightarrow H)^0$	$\{\{M(G^1)\},\{M(H^0)\}\}$

type	F^{pol}	$M(F^{pol})$
β	$(G \wedge H)^0$	$\{\{M(G^0),M(H^0)\}\}$
	$(G \vee H)^1$	$\{\{M(G^1),M(H^1)\}\}$
	$(G \Rightarrow H)^1$	$\{\{M(G^0),M(H^1)\}\}$
γ	$(\forall xG)^1$	$M(G[x\backslash x^*]^1)$
	$(\exists xG)^0$	$M(G[x\backslash x^*]^0)$
δ	$(\forall xG)^0$	$M(G[x\backslash t^*]^0)$
	$(\exists xG)^1$	$M(G[x\backslash t^*]^1)$

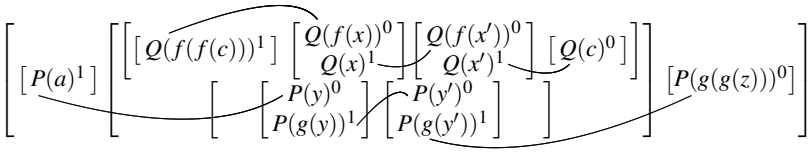

Fig. 1. Graphical representation of a non-clausal matrix and its non-clausal connection proof

The non-clausal connection calculus for classical logic is *sound* and *complete* [15]. It has the same axiom, start rule, and reduction rule as the formal *clausal* connection calculus [17]. The extension rule is slightly modified and a decomposition rule is added. A few additional concepts are required as follows in order to specify which clauses C_1 can be used within the non-clausal extension rule. See [15] for details and examples.

A clause C *contains* a literal L (or clause C'') iff $L \in C$ or C' contains L ($C = C''$ or C' contains C'') for a matrix $M' \in C$ with $C' \in M$. A clause C is α*-related* to a literal L iff it occurs besides L in the graphical matrix representation, i.e., $\{C', C''\} \subseteq M'$ for a matrix M' such that C' contains L and C'' contains C. In the *copy of a clause* C all *free variables* in C are replaced by fresh variables. $M[C_1 \backslash C_2]$ denotes the matrix M, in which the clause C_1 is replaced by the clause C_2. C' is a *parent clause* of C iff $M' \in C'$ and $C \in M'$ for some matrix M'. 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 α-related to all literals of $Path$ occurring in M and if C has a parent clause, it contains a literal of $Path$. In the β*-clause of* C_2 *with respect to* L_2, denoted by $\beta\text{-}clause_{L_2}(C_2)$, L_2 and all clauses that are α-related to L_2 are deleted from C_2 (in the new subgoal C_3).

The analytic, i.e., bottom-up *proof search* in the non-clausal calculus is carried out in the same way as in the clausal calculus. Additional *backtracking* might be required when choosing C_1 in the decomposition rule; no backtracking is required when choosing M_1. The *rigid* term substitution σ is calculated whenever a connection is identified in an application of the reduction or extension rule. On formulae in clausal form, the non-clausal connection calculus coincides with the clausal connection calculus. *Optimization techniques*, such as positive start clauses, regularity, lemmata and restricted backtracking, can be employed in a way similar to the clausal connection calculus [14].

Axiom (A)	$\dfrac{}{\{\}, M, Path}$	
Start (S)	$\dfrac{C_2, M, \{\}}{\varepsilon, M, \varepsilon}$	and C_2 is copy of $C_1 \in M$
Reduction (R)	$\dfrac{C, M, Path \cup \{L_2\}}{C \cup \{L_1\}, M, Path \cup \{L_2\}}$	and $\sigma(L_1) = \sigma(\overline{L_2})$
Extension (E)	$\dfrac{C_3, M[C_1 \backslash C_2], Path \cup \{L_1\} \quad C, M, Path}{C \cup \{L_1\}, M, Path}$	and $C_3 := \beta\text{-}clause_{L_2}(C_2)$, C_2 is copy of C_1, C_1 is e-clause of M wrt. $Path \cup \{L_1\}$, C_2 contains L_2 with $\sigma(L_1) = \sigma(\overline{L_2})$
Decomposition (D)	$\dfrac{C \cup C_1, M, Path}{C \cup \{M_1\}, M, Path}$	and $C_1 \in M_1$

Fig. 2. The non-clausal connection calculus

3 The Implementation

The implementation of the non-clausal connection calculus of Fig. 2 follows the *lean methodology* [3], which is already used for the clausal connection prover leanCoP [17]. 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 *na*tural *no*nclausal *co*nnection *p*rover nanoCoP is available under the GNU General Public License and can be downloaded at `http://www.leancop.de/nanocop/`.

Non-clausal Matrices. In a first step the input formula F is translated into a non-clausal (indexed) matrix $M(F)$ according to Table 1; redundant brackets of the form "$\{\{\ldots\}\}$" are removed [15]. Additionally, every (sub-)clause $(I,V):C$ and (sub-)matrix $J:M$ is marked with a unique *index* I and 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 atoms, 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 $M_{\#}$ from Sec. 2 is represented by the Prolog term

```
[(1^K)^[]:[-(p(a))],
 (2^K)^[]:[3^K:[(4^K)^[]:[-(q(f(f(c))))],(5^K)^[X]:[q(f(X)), -(q(X))],
                (6^K)^[]:[q(c)]], 7^K:[(8^K)^[Y]:[p(Y), -(p(g(Y)))]]],
 (9^K)^[Z]:[p(g(g(Z)))]]
```

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,ClaB,ClaC,Grnd)
```

is asserted into the database where `ClaC` $\in M$ is the (largest) clause in which `Lit` occurs and `ClaB` is the β-clause of `ClaC` with respect to `Lit`. `Grnd` is set to `g` if the smallest clause in which `Lit` occurs is ground, i.e. does not contain any variables; otherwise `Grnd` is set to `n`. Storing literals of M in the database in this way is called *lean Prolog technology* [14] and integrates the advantages of the Prolog technology approach [23] into the lean theorem proving framework. No other modifications or simplifications of the original formula (structure) are done during these two preprocessing steps.

Non-clausal Proof Search. The nanoCoP source code is shown in Fig. 3. It uses only the standard Prolog predicates `member`, `append`, `length`, `assert`, `retract`, `copy_term`, `unify_with_occurs_check`, and the additional predicate `positiveC(Cla,Cla1)`, which returns the clause `Cla1` in which all clauses that are not positive in `Cla` are deleted. A clause is positive if all of its elements (matrices and literals) are positive; a matrix is positive if it contains at least one positive clause; a literal is positive if its polarity is 0.

The predicate `prove(Mat,PathLim,Set,Proof)` implements the start rule (lines *1–8*). `Mat` is the matrix generated in the preprocessing step. `PathLim` is the maximum size of the active path used for iterative deepening, `Set` is a list of options used to control the proof search, and `Proof` contains the returned connection proof. Start clauses are restricted to positive clauses (line *2*) before the actual proof search is invoked (line *3*). If no proof is found with the current active path limit `PathLim` and this limit was reached, then `PathLim` is increased and the proof search starts over again (lines *4–8*).

The predicate `prove(Cla,Mat,Path,PathI,PathLim,Lem,Set,Proof)` implements the axiom (line 9), the decomposition rule (lines 10–14), the reduction rule (lines 15–18, 21–22, 31), and the extension rule (lines 15–18, 24–42) of the non-clausal connection calculus in Fig. 2. `Cla`, `Mat`, and `Path` represent the subgoal clause C, the (indexed) matrix M and the (active) *Path*. The *indexed path* `PathI` 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 been "solved" already [14]. `Set` is a list of options and may contain the elements "`cut`" and "`comp(`I`)`" for $I \in \mathbb{N}$, which are used to control the restricted backtracking technique [14]. This `prove` predicate succeeds iff there is a connection proof for the tuple (`Cla`, `Mat`, `Path`) with $|\texttt{Path}| < \texttt{PathLim}$. In this case `Proof` contains a compact connection proof. The input matrix `Mat` has to be stored in Prolog's database (as explained above).

When the decomposition rule is applied, a clause `Cla1` of the first matrix of the subgoal clause `[J:Mat|Cla]` is selected (line 11). The search continues with clause `Cla1` (line 12) using the extended indexed path `[I,J|PI]`, and the remaining elements of `Cla` (line 13). For the reduction and extension rules the complement `NegLit` of the first literal `Lit` of the subgoal clause is calculated (line 18) and used for the following reduction and extension step. When the reduction rule is applied, it is checked whether the active `Path` contains a literal `NegL` that unifies with `NegLit` (line 21). In this case the proof search continues with the clause `Cla` for the premise of the reduction rule (line 31). When the extension rule is applied, the predicate `lit(NegLit,ClaB,Cla,Grnd1)` is called to find a clause in Prolog's database that contains the complement `NegLit` of the literal `Lit` (line 24). For this operation sound unification has to be switched on (in, e.g., ECLiPSe Prolog this is done by calling "`set_flag(occur_check,on)`" before the proof search starts). The predicate `prove_ec` calculates an appropriate extension clause and returns its β-clause `ClaB1` with respect to `NegLit` (line 27). The proof search continues with `ClaB1` as new subgoal clause for the left premise of the extension rule with the literal `Lit` added to the active `Path` (line 28), and with the remaining subgoal clause `Cla` for the right premise (line 31). The substitution σ is stored implicitly by Prolog.

The predicate `prove_ec(ClaB,Cla1,Mat,ClaB1,Mat1)` is used to calculate extension clauses (lines 32–42). Starting with the (largest possible) extension clause `Cla1`, its β-clause `ClaB`, and the current (indexed) matrix `Mat`, this predicate returns an appropriate extension clause `Cla`, copies it into `Mat` and returns its β-clause `ClaB1` and the new matrix `Mat1`. The extension clause has to fulfil the conditions described in Sec. 2: it has to be (a) large enough to contain a literal of `Path` or (b) small enough to be α-related to all literals of `Path` occurring in `Mat` and again large enough that in case it has a parent clause, this contains a literal of `Path`; in both cases the extension clause has to be large enough such that the literal `Lit` unifies with the literal `NegLit` in the current matrix. As an optimization only extension clauses that introduce new variables are considered.

Prolog depth-first search results in an incomplete proof search. In order to regain completeness nanoCoP performs an *iterative deepening* on the size of the active path. When the extension rule is applied and the extension clause is not ground, it is checked whether the size `K` of the active `Path` exceeds the current path limit `PathLim` (line 25). In this case the predicate `pathlim` is written into Prolog's database (line 26) indicating the need to increase the path limit if the proof search fails for the current path limit.

nanoCoP uses additional optimization techniques that are already used in the classical (clausal) connection prover leanCoP [14]: *regularity* (line 17), *lemmata* (line 19), and *restricted backtracking* (line 30). *Regularity* ensures that no literal occurs more than once in the active path. The idea of *lemmata* (or factorization) is to reuse subproofs during the proof search. *Restricted backtracking* is a very effective technique for pruning the search space in connection calculi [14]. It is switched on if the list `Set` contains the element "`cut`". If it also contains "`comp(`I`)`" for $I \in \mathbb{N}$, then the proof search restarts again without restricted backtracking if the path limit `PathLim` exceeds I.

```
        % start rule
(1)     prove(Mat,PathLim,Set,[(I^0)^V:Cla1|Proof]) :-
(2)         member((I^0)^V:Cla,Mat), positiveC(Cla,Cla1), Cla1\=!,
(3)         prove(Cla1,Mat,[],[I^0],PathLim,[],Set,Proof).
(4)     prove(Mat,PathLim,Set,Proof) :-
(5)         retract(pathlim) ->
(6)         ( member(comp(PathLim),Set) -> prove(Mat,1,[],Proof) ;
(7)           PathLim1 is PathLim+1, prove(Mat,PathLim1,Set,Proof) ) ;
(8)         member(comp(_),Set) -> prove(Mat,1,[],Proof).

        % axiom
(9)     prove([],_,_,_,_,_,_,[]).

        % decomposition rule
(10)    prove([J:Mat1|Cla],MI,Path,PI,PathLim,Lem,Set,Proof) :- !,
(11)        member(I^_:Cla1,Mat1),
(12)        prove(Cla1,MI,Path,[I,J|PI],PathLim,Lem,Set,Proof1),
(13)        prove(Cla,MI,Path,PI,PathLim,Lem,Set,Proof2),
(14)        append(Proof1,Proof2,Proof).

        % reduction and extension rules
(15)    prove([Lit|Cla],MI,Path,PI,PathLim,Lem,Set,Proof) :-
(16)        Proof=[[I^V:[NegLit|ClaB1]|Proof1]|Proof2], copy_term(Lit,LitV),
(17)        \+ (member(LitC,[Lit|Cla]), member(LitP,Path), LitC==LitP),
(18)        (-NegLit=Lit;-Lit=NegLit) ->
(19)           ( member(LitL,Lem), Lit==LitL, ClaB1=[], Proof1=[]
(20)             ;
(21)             member(NegL,Path), unify_with_occurs_check(NegL,NegLit),
(22)             ClaB1=[], Proof1=[]
(23)             ;
(24)             lit(NegLit,ClaB,Cla1,Grnd1),
(25)             ( Grnd1=g -> true ; length(Path,K), K<PathLim -> true ;
(26)               \+ pathlim -> assert(pathlim), fail ),
(27)             prove_ec(ClaB,Cla1,MI,PI,I^V:ClaB1,MI1),
(28)             prove(ClaB1,MI1,[Lit|Path],[I|PI],PathLim,Lem,Set,Proof1)
(29)           ),
(30)           ( (member(cut,Set);Lit==LitV) -> ! ; true ),
(31)           prove(Cla,MI,Path,PI,PathLim,[Lit|Lem],Set,Proof2).

        % extension clause (e-clause)
(32)    prove_ec((I^K)^V:ClaB,IV:Cla,MI,PI,ClaB1,MI1) :-
(33)        append(MIA,[(I^K1)^V1:Cla1|MIB],MI), length(PI,K),
(34)        ( ClaB=[J^K:[ClaB2]|_], member(J^K1,PI),
(35)          unify_with_occurs_check(V,V1), Cla=[_:[Cla2|_]|_],
(36)          append(ClaD,[J^K1:MI2|ClaE],Cla1),
(37)          prove_ec(ClaB2,Cla2,MI2,PI,ClaB1,MI3),
(38)          append(ClaD,[J^K1:MI3|ClaE],Cla3),
(39)          append(MIA,[(I^K1)^V1:Cla3|MIB],MI1)
(40)          ;
(41)          (\+member(I^K1,PI);V\==V1;V\=[]) ->
(42)          ClaB1=(I^K)^V:ClaB, append(MIA,[IV:Cla|MIB],MI1) ).
```

Fig. 3. Source code of the nanoCoP prover

4 Experimental Evaluation

These 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, and a CPU time limit of 100 seconds.

The following formula F_n is a slightly extended version of the formula $F_\#$ in Sec. 2, where f^n, g^n, h^n, and i^n are abbreviations for n nested applications of these functions:

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

Table 2 shows the results on this (valid) formula class for n=10, n=30, and n=90 for the following provers: the lean (non-clausal) tableau prover leanTAP [3], the resolution prover Prover9 [12], the superposition prover E [22] (using options "`--auto --tptp3-format`"), leanCoP [14, 17], 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=[]`). Times are given in seconds; "size" is the number of nodes in the returned proof tree.

Table 2. Results on formula class F_n

n=		leanTAP 2.3	Prover9 2009-02A	E 1.9	– leanCoP 2.2 – [nodef]	[def]	nanoCoP [ ]
10	time (size)	0.17 (128)	–	1.22 (2916)	–	–	**0.09 (45)**
30	time (size)	–	–	84.57 (57628)	–	–	**0.12 (125)**
90	time (size)	–	–	–	–	–	**0.42 (365)**

Table 3 shows the test results on all 5051 first-order (FOF) problems in the TPTP library v3.7.0 [24]. For leanTAP, leanCoP, and nanoCoP, the required equality axioms were added in a preprocessing step (which is included in the timings). The full leanCoP prover ("full") additionally uses strategy scheduling [14]. For nanoCoP, a restricted backtracking strategy, i.e. `Set=[cut,comp(6)]`, was tested as well. The nanoCoP core prover perform significantly better than both clausal form translations of the leanCoP core prover. 40%/51% of the proofs found by nanoCoP (without restricted backtracking) are shorter than those of leanCoP [nodef]/[def], respectively; as many of these problems are (mostly) in clausal form, 56%/47% of the proofs have the same size. The nanoCoP proofs are up to 96%/74% shorter than those of leanCoP [nodef]/[def], respectively. The classical version of the non-clausal connection prover JProver [21] has a lower performance than leanTAP (also reflected in its intuitionistic performance [13]).

Table 3. Results on TPTP library v3.7.0

	leanTAP 2.3	Prover9 2009-02A	E 1.9	—- leanCoP 2.2 —- [nodef]	[def]	"full"	—- nanoCoP 1.0 —- [ ]	[cut,comp(6)]
proved	404	1611	2782	1134	1065	1710	**1232**	**1485**
0 to 1sec.	379	1285	2104	938	865	1215	**1001**	**1172**
1 to 10sec.	13	200	338	113	125	216	**139**	**157**
10 to 100sec.	4	126	340	83	75	279	**92**	**156**

5 Conclusion

This paper presents nanoCoP, a non-clausal connection prover for classical first-order logic. Using non-clausal matrices the proof search works directly on the original structure of the input formula. No translation steps to any clausal or other normal form are required. This combines the advantages of more *natural* non-clausal tableau or sequent 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. It is expected that the integration of strategy scheduling will also outperform the "full" leanCoP prover. About half of the returned non-clausal proofs are up to 96% shorter than their clausal counterparts. By using the standard translation, i.e. applying the distributive laws, the size of the resulting formula might grow exponentially with respect to the size of the original formula, which is not feasible for some formulae. The definitional translation [19] introduces definitions for subformulae, which results in a significant overhead for the proof search as well [14]. Other clausal form translations that work well for resolution provers, e.g. the ones used in E or Flotter, have a significant lower performance when used in with a connection prover [14].

Both clausal form translations modify the structure of the original formula, which makes it difficult to translate the (clausal) proof back into a proof of the original formula [20]. nanoCoP returns a compact non-clausal connection proof, which directly represents a free-variable tableau proof. A connection in the nanoCoP proof corresponds to a closed branch in the tableau calculus [7] or an axiom in the sequent calculus [6]. The translation into, e.g., a sequent proof is straightforward, when skolemization is seen as a way to encode the eigenvariable condition of the sequent calculus. This close relationship to the sequent calculus makes nanoCoP an ideal tool to be used within interactive proof systems, such as Coq, Isabelle, HOL or NuPRL. The compact size of nanoCoP makes it also a suitable tool for the development of verifiably correct software [18], as its correctness can be proven much more easily than that of a large proof system consisting of several thousand lines of source code.

Only few research work investigates non-clausal connection calculi and their implementations. Other non-clausal calculi [1, 5, 8, 11] work only on ground formulae. For first-order formulae, copies of subformulae have to be added iteratively, which introduces a huge redundancy into the proof search, as already observed with JProver [21]. For an efficient proof search, clauses have to be added dynamically *during* the proof search. Some older non-clausal implementations [9] are not available anymore.

Another important application of nanoCoP is its usage within non-classical logics, such as intuitionistic or modal *first-order* 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 [13, 16]. In order to improve performance, further optimization techniques need to be integrated into nanoCoP, such as strategy scheduling [14], learning [10] or variable splitting [2].

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

References

1. Andrews, P. B.: Theorem proving via general matings. Journal of the ACM 28, 193–214 (1981)
2. Antonsen, R., Waaler, A.: Liberalized variable splitting. Journal of Automated Reasoning 38, 3–30 (2007)
3. Beckert, B., Posegga, J.: leanTAP: lean, tableau-based deduction. Journal of Automated Reasoning 15(3), 339–358 (1995)
4. Bibel, W.: Matings in matrices. Communications of the ACM 26, 844–852 (1983)
5. Bibel, W.: Automated Theorem Proving. 2nd edition. Vieweg, Wiesbaden (1987)
6. Gentzen, G.: Untersuchungen über das logische Schließen. Mathematische Zeitschrift 39, 176–210, 405–431 (1935)
7. Hähnle, R.: Tableaux and Related Methods. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 100–178. Elsevier, Amsterdam (2001)
8. Hähnle, R., Murray, N.V., Rosenthal, E.: Linearity and Regularity with Negation Normal Form. Theoretical Computer Science 328, 325–354 (2004)
9. Issar, S.: Path-Focused Duplication: A Search Procedure for General Matings. AAAI-90 Proceedings, pp. 221–226 (1990)
10. Kaliszyk, C., Urban, J.: FEMaLeCoP: Fairly Efficient Machine Learning Connection Prover. In: Davis, M. et al. (eds.) LPAR 2015. LNAI, vol. 9450, pp. 88–96. Springer, Heidelberg (2015)
11. Kreitz, C., Otten, J.: Connection-based theorem proving in classical and non-classical logics. Journal of Universal Computer Science 5, 88–112 (1999)
12. McCune, W.: Release of Prover9. Mile high conference on quasigroups, loops and nonassociative systems, Denver (2005)
13. Otten, J.: leanCoP 2.0 and ileanCoP 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic. In: Armando, A. et al. (eds.) IJCAR 2008, LNAI, vol. 5195, pp. 283–291. Springer, Heidelberg (2008)
14. Otten, J.: Restricting backtracking in connection calculi. AI Communications 23, 159–182 (2010)
15. Otten, J.: A Non-clausal Connection Calculus. In: Brünnler, K., Metcalfe, G. (eds.) TABLEAUX 2011, LNAI, vol. 6793, pp. 226–241. Springer, Heidelberg (2011)
16. Otten, J.: MleanCoP: A Connection Prover for First-Order Modal Logic. In: Demri, S. et al. (eds.) IJCAR 2014, LNAI, vol. 8562, pp. 269–276. Springer, Heidelberg (2014)
17. Otten, J., Bibel, W.: leanCoP: lean connection-based theorem proving. Journal of Symbolic Computation 36, 139–161 (2003)
18. Otten, J., Bibel, W.: Advances in Connection-Based Automated Theorem Proving. In: Bowen, J. at al. (eds.) Provably Correct Systems. Springer, Heidelberg (to appear)
19. Plaisted, D., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation 2, 293–304 (1986)
20. Reis, G.: Importing SMT and connection proofs as expansion trees. In: Kaliszyk, C., Paskevich, A. (eds.) 4th Workshop on Proof eXchange for Theorem Proving (PxTP15), EPTCS 186, pp. 3–10 (2015)
21. Schmitt, S. et al.: JProver: Integrating Connection-based Theorem Proving into Interactive Proof Assistants. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001, LNAI, vol. 2083, pp. 421–426. Springer, Heidelberg (2001)
22. Schulz, S.: E – a brainiac theorem prover. AI Communications 15(2), 111–126 (2002)
23. Stickel, M.: A Prolog technology theorem prover: implementation by an extended Prolog compiler. Journal of Automated Reasoning 4, 353–380 (1988)
24. Sutcliffe, G.: The TPTP problem library and associated infrastructure: the FOF and CNF parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)