Proof Search Optimizations for Non-clausal Connection Calculi

Jens Otten

Department of Informatics, University of Oslo
PO Box 1080 Blindern, 0316 Oslo, Norway
jeotten@ifi.uio.no

Abstract. The paper presents several proof search optimization techniques for non-clausal connection calculi. These techniques are implemented and integrated into the non-clausal connection prover nanoCoP. Their effectiveness is evaluated on the problems in the TPTP library. Together with a fixed strategy scheduling, these techniques are the basis of the new version 1.1 of nanoCoP.

1 Introduction

Most of the popular efficient proof search calculi require the input formula to be in clausal form, i.e. in disjunctive or conjunctive normal form. First-order formulae that are not in clausal 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. Even the definitional translation into clausal form introduces a significant overhead into the proof search [7]. 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 [14].

Non-clausal connection calculi for classical logic [8] and non-classical logics [11] do not have these shortcomings. They combine the advantages of more *natural* non-clausal calculi, such as sequent or (standard) tableau calculi [2, 3], with the *efficiency* of a connection-based proof search. Recently, implementations of these calculi have been presented: nanoCoP for classical logic [9, 10] and nanoCoP-i/nanoCoP-M for intuitionistic/modal first-order logic [11]. But whereas the nanoCoP core provers have a better performance than the core provers of the leanCoP series [9, 11], the full leanCoP provers include additional optimizations techniques and a strategy scheduling, which leads to a significantly better performance than the current nanoCoP (core) provers.

This paper presents techniques for optimizing the proof search in *non-clausal* connection calculi (Section 3) and shows how these can be implemented and integrated into the (classical) nanoCoP prover (Section 4). Among these techniques are restricted backtracking and clause reordering techniques, which are already successfully used in clausal connection calculi [7]. Using these techniques within a fixed strategy scheduling, results in a new version 1.1 of nanoCoP. The effectiveness of all presented optimization techniques is evaluated and the nanoCoP 1.1 prover is compared to other popular provers on the problems in the TPTP library (Section 5).

2 The Non-clausal Connection Calculus

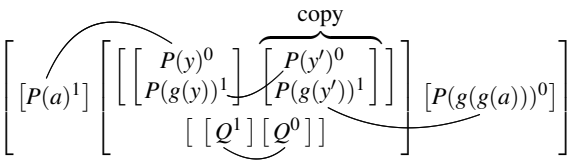
A *(first-order) formula* (denoted by F) is built up from atomic formulae, the connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and the standard first-order quantifiers $\forall$ and $\exists$. An *atomic formula* (denoted by A) is built up from predicate symbols and terms. A *literal* L has the 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 polarities. A *term substitution* σ assigns terms to variables. The non-clausal connection calculus uses non-clausal matrices. A *non-clausal matrix* M of a formula F is a set of clauses, in which each clause consists of literals *and* (sub)matrices, and can be seen as a representation of a formula in negation normal form; see [8] for details. In the *graphical representation* of a non-clausal matrix, its clauses are arranged horizontally, while the literals and matrices of each clause are arranged vertically.

The axiom and the rules of the *non-clausal connection calculus* are given in Figure 1. Compared to the formal *clausal* connection calculus [12], a decomposition rule for decomposing clauses is added and the extension rule with its extension clause (e-clause) is generalized; see [8, 9] for details. 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 "through M" or ε; σ is a (rigid) term substitution. A *non-clausal connection proof* of M is a non-clausal connection proof of $\varepsilon, M, \varepsilon$. The *rigid* term substitution σ is calculated whenever a connection is identified, i.e. a reduction or extension rule is applied.

For example, the formula $P(a) \wedge (\forall y(P(y) \Rightarrow P(g(y))) \vee \neg(Q \Rightarrow Q)) \Rightarrow P(g(g(a)))$ has the non-clausal matrix (the *polarity* 1 is used to represent negation):

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

which has the following graphical representation and connection proof (using the term substitution σ with $\sigma(y) = a$ and $\sigma(y') = g(a)$):



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. 1. The non-clausal connection calculus

3 Optimization Techniques

The following proof search optimization techniques are integrated into the non-clausal connection calculus: regularity, lemmata, restricted backtracking, positive and conjecture start clauses, reordering clauses, and strategy scheduling.

3.1 Regularity and Lemmata

Regularity is an effective technique for pruning the search space in *clausal* connection calculi [5]. The *regularity condition* states that no literal occurs more than once in the active path (without losing completeness). This condition can be integrated in a similar way[1] into the non-clausal connection calculus in Figure 1 by adding the following restriction to the reduction rule and the extension rule:

$$\forall\ L' \in C \cup \{L_1\} :\ \sigma(L') \notin \sigma(Path)\ .$$

Additional backtracking is avoided if the term substitution σ is not modified in order to satisfy the regularity condition, e.g., by applying the restriction only to ground literals.

The idea of *lemmata* is to reuse subproofs during the proof search [5]. To this end an additional set of lemmata (i.e. set of literals) and a lemma rule is added to the non-clausal connection calculus in the same way as it is added to the clausal calculus [7].

3.2 Restricted Backtracking

As proof search in the presented connection calculus is *not* confluent, *backtracking* is necessary in order to achieve completeness. In the *clausal* connection calculus backtracking is necessary when choosing clause C_1 in the extension rule or literal L_1 in the reduction or extension rule. In the non-clausal calculus (see Figure 1) *additional* backtracking is required when choosing C_1 in the decomposition rule (but *no* backtracking is required when choosing M_1).

The idea of *restricted backtracking* is to cut off any alternative solutions/connections once a literal from the subgoal clause has been solved [7]. A literal L is called *solved* if it is the literal L_1 of a reduction or extension step (see Figure 1) and in case of the extension rule, there is a proof for the left premise. Furthermore, backtracking can also be restricted when selecting clause C_1 in the start rule by omitting alternative start clauses. Restricted backtracking is correct (as the search space is only pruned) and very effective, but incomplete [7]. It can also be applied to the non-clausal calculus.

3.3 Positive and Conjecture Start Clauses

By default, the selection of the start clause C_1 in the start rule (see Figure 1) is restricted to positive clauses. A clause is *positive* iff *all* of its elements (matrices and literals) are positive; a matrix is positive iff it contains at least one positive clause; a literal is positive iff its polarity is 0. Furthermore, clauses that are *not* positive can be deleted from the start clause C_1 (or its copy C_2).

[1] Observe that the restriction to the *active* path is necessary in the *non-clausal* case as completeness is lost, otherwise; see, e.g., the valid matrix $\{\{p^0\},\{p^1,\{\{p^0\},\{q^0\}\}\},\{q^1\}\}$.

Whereas selecting positive start clauses can significantly prune the search space, this technique is less appropriate for problems with a large number of axioms, i.e. formulae of the form $A_1 \wedge ... \wedge A_n \Rightarrow C$ for a "large" n. In this case it is more goal-oriented to start with clauses of the *conjecture* C, which leads to a smaller search space.[2]

3.4 Reordering Clauses

As already mentioned in Section 3.2, the connection calculus is not proof confluent. The order in which clauses and literals are selected if more than one connection is possible has a significant impact on the proof search. This is even more important for incomplete techniques, such as restricted backtracking. One clause order might lead to an incomplete proof search, while another one might quickly lead to a proof. Hence, *reordering clauses* is a crucial technique for connection calculi using restricted backtracking. For non-clausal calculi it is important that a reordering technique leads to diverse orders even for small sets of clauses, e.g., if a (sub)matrix contains only two or three clauses.

3.5 Strategy Scheduling

Trying different techniques or methods when solving a hard problem is, in general, a successful approach. Hence, when trying to prove a formula, a mix of different techniques or strategies is more likely to be successful. A *strategy scheduling* would subsequently try different techniques to find a proof of a formula, e.g., positive and conjecture start clauses, with and without restricted backtracking, or using different clause orders.

4 Implementation

The proof search optimization techniques described in Section 3 have been implemented and integrated into the non-clausal connection prover nanoCoP, which is available under the GNU General Public License and can be downloaded from the following website: `http://www.leancop.de/nanocop/`.

In a first step the input formula F is translated into a non-clausal (indexed) matrix M; in the second step this matrix is written into Prolog's database using the literal `lit(Lit,ClaB,ClaC,Grnd)`; see [9] for details.

The source code of the nanoCoP core prover is shown in Figure 2. The predicate `prove(Mat,PathLim,Set,Proof)` implements the start rule (lines 1–11). `Mat` is the matrix generated in the preprocessing step, `PathLim` is the maximum size of the active path used for iterative deepening (lines 7–11), `Set` is a strategy (see below), and `Proof` contains the returned connection proof.

The predicate `prove(Cla,Mat,Path,PathI,PathLim,Lem,Set,Proof)` implements the axiom (line 12), the decomposition rule (lines 13–17), the reduction rule (lines 18–21, 24–25, 34), and the extension rule (lines 18–21, 27–45) of the non-clausal connection calculus in Figure 1. It succeeds iff there is a connection proof for the tuple "`Cla`, `Mat`, `Path`" with $|\texttt{Path}| < \texttt{PathLim}$. The predicate `prove_ec` calculates an appropriate extension clause (lines 35–45). The substitution σ is stored implicitly by Prolog.

[2] This approach is incomplete, though, as shown by the following formula: $(P \wedge \neg P) \Rightarrow Q$.

```
        % start rule
(1)     prove(Mat,PathLim,Set,[(I^0)^V:Cla1|Proof]) :-
(2)         ( member(scut,Set) -> ( append([(I^0)^V:Cla1|_],[!|_],Mat) ;
(3)             member((I^0)^V:Cla,Mat), positiveC(Cla,Cla1) ) -> true ;
(4)             ( append(MatC,[!|_],Mat) -> member((I^0)^V:Cla1,MatC) ;
(5)             member((I^0)^V:Cla,Mat), positiveC(Cla,Cla1) ) ),
(6)         prove(Cla1,Mat,[],[I^0],PathLim,[],Set,Proof).

(7)     prove(Mat,PathLim,Set,Proof) :-
(8)         retract(pathlim) ->
(9)         ( member(comp(PathLim),Set) -> prove(Mat,1,[],Proof) ;
(10)          PathLim1 is PathLim+1, prove(Mat,PathLim1,Set,Proof) ) ;
(11)        member(comp(_),Set) -> prove(Mat,1,[],Proof).

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

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

        % reduction and extension rules
(18)    prove([Lit|Cla],MI,Path,PI,PathLim,Lem,Set,Proof) :-
(19)        Proof=[[I^V:[NegLit|ClaB1]|Proof1]|Proof2],
(20)        \+ (member(LitC,[Lit|Cla]), member(LitP,Path), LitC==LitP),
(21)        (-NegLit=Lit;-Lit=NegLit) ->
(22)           ( member(LitL,Lem), Lit==LitL, ClaB1=[], Proof1=[]
(23)             ;
(24)             member(NegL,Path), unify_with_occurs_check(NegL,NegLit),
(25)             ClaB1=[], Proof1=[]
(26)             ;
(27)             lit(NegLit,ClaB,Cla1,Grnd1),
(28)             ( Grnd1=g -> true ; length(Path,K), K<PathLim -> true ;
(29)               \+ pathlim -> assert(pathlim), fail ),
(30)             prove_ec(ClaB,Cla1,MI,PI,I^V:ClaB1,MI1),
(31)             prove(ClaB1,MI1,[Lit|Path],[I|PI],PathLim,Lem,Set,Proof1)
(32)           ),
(33)           ( member(cut,Set) -> ! ; true ),
(34)           prove(Cla,MI,Path,PI,PathLim,[Lit|Lem],Set,Proof2).

        % extension clause (e-clause)
(35)    prove_ec((I^K)^V:ClaB,IV:Cla,MI,PI,ClaB1,MI1) :-
(36)        append(MIA,[(I^K1)^V1:Cla1|MIB],MI), length(PI,K),
(37)        ( ClaB=[J^K:[ClaB2]|_], member(J^K1,PI),
(38)          unify_with_occurs_check(V,V1), Cla=[_:[Cla2|_]|_],
(39)          append(ClaD,[J^K1:MI2|ClaE],Cla1),
(40)          prove_ec(ClaB2,Cla2,MI2,PI,ClaB1,MI3),
(41)          append(ClaD,[J^K1:MI3|ClaE],Cla3),
(42)          append(MIA,[(I^K1)^V1:Cla3|MIB],MI1)
(43)          ;
(44)          (\+member(I^K1,PI);V\==V1) ->
(45)          ClaB1=(I^K)^V:ClaB, append(MIA,[IV:Cla|MIB],MI1) ).
```

Fig. 2. Source code of the nanoCoP 1.1 core prover

Regularity and Lemmata The *regularity* condition is implemented in line 20. In order to avoid backtracking, it is restricted to ground literals, i.e. literals without term variables or literals whose term variables have been substituted by terms that do not contain term variables. The *lemma rule* is implemented in line 22. Furthermore, the list `Lem` containing all literals that have been "solved" so far is added to the arguments of the `prove` predicate. In line 34 the current literal `Lit` is added as lemma to the set `Lem`.

Restricted Backtracking The restricted backtracking technique for start clauses is implemented in line 2 and 3. Restricted backtracking for the reduction and extension rule (and the lemma rule) is implemented in line 33. In the former case an implicit Prolog cut in an if-then-else condition is used to cut off alternative start clauses, in the latter case a Prolog cut is used to cut off alternative connections. In both cases, restricted backtracking can be switched on or off (see below for details).

Positive and Conjecture Start Clauses By default, start clauses are restricted to *positive* clauses (lines 3 and 5). The predicate `positiveC(Cla,Cla1)` returns a positive start clause `Cla1` if clause `Cla` is positive. It is implemented within another seven lines of Prolog code. If start clauses are restricted to *conjecture* clauses, a "`!`" is inserted into the matrix to split the clauses of the conjecture formula from the axiom clauses. In this case only clauses in front of the "`!`" are considered as start clauses (lines 2 and 4). See the full source code on the nanoCoP website for details.

Reordering Clauses By default, clauses are arranged in ascending order according to the number of paths through its elements. If reordering of clauses is switched on (see Section 4), all (sub-)clauses are reordered using a compact shuffle algorithm. It is implemented within 13 lines of Prolog code; see the full source code for details.

Strategy Scheduling The argument `Set` of the `prove` predicate (see, e.g., lines 1, 7, 13, and 18) defines a *strategy*. It is a list of options and may contain the following elements:

- "`scut`" (switch on restricted backtracking for start clauses),
- "`cut`" (switch on restricted backtracking for reduction, extension and lemma rule),
- "`conj`" (use conjecture start clauses),
- "`reo(`J`)`" for $J \in \mathbb{N}$ (reorder the clauses J times before the proof search starts), and
- "`comp(`I`)`" for $I \in \mathbb{N}$ (restart proof search using a complete search strategy, i.e. without `scut`, `cut`, and `conj`, if `PathLim` exceeds I).

These are the main strategies used for the *fixed strategy scheduling* together with the total time ratio allotted to them: `[cut,comp(7)]`/15%, `[reo(22),conj,cut]`/20%, `[scut]`/10%, `[scut,cut]`/5%, `[reo(20),conj,cut]`/10%, `[reo(30),conj,scut,cut]`/10%, and `[reo(34),conj,scut,cut]`/5%. Afterwards, 12 different strategies are invoked for 20% of the total time. Finally, the strategy `[]` is used for the remaining time ($\geq$ 5%). As the first strategy and the last strategy are complete, the whole prover is complete (provided an arbitrary large total time is given). The strategy scheduling is implemented with a *shell script* that invokes the Prolog prover.

5 Experimental Evaluation

The different optimization techniques described in Section 3 and implemented in Section 4 were evaluated on all 8044 first-order (so-called FOF) problems in the TPTP library v6.4.0 [16]. Furthermore, nanoCoP was compared to other popular provers. All evaluations were conducted on a 2.3 GHz Xeon system with 32 GB of RAM running Linux 2.6.32. ECLiPSe Prolog 5.10 was used for all provers implemented in Prolog.

5.1 Evaluation of Different Optimizations

The following versions of nanoCoP are evaluated: a version without regularity and lemmata ("basic"), the standard version using regularity and lemmata (i.e. strategy `[]`), a version with conjecture start clauses (`[conj]`), two versions with restricted backtracking (`[scut]` and `[cut]`), nanoCoP 1.0, which uses only the strategy `[cut,comp(7)]`, a version with reordering clauses ("reo"=`[reo(22),conj,cut]`), and nanoCoP 1.1 using the strategy scheduling described in Section 4.

Table 1 shows the results of the evaluation for a CPU time limit of 10 seconds. The table shows the number of proved (valid) problems (i.e. with TPTP status "Theorem" or "Unsatisfiable") and the number of refuted (invalid) problems (i.e. with TPTP status "Countersatisfiable" or "Satisfiable"). The entry "errors" shows the number of problems for which nanoCoP terminates with a stack overflow error by Prolog. It also shows the number of new problems ("new proved") that were not proved by a preceding version ("compared to") of nanoCoP.

Whereas regularity and lemmata (`[]`) show only a small improvement, conjecture start clauses (`[conj]`) and restricted backtracking (`[scut]` and `[cut]`) are very effective techniques proving many more problems. For example, the `[cut]` version proves 421 new problems compared to the standard version (`[]`). Using the strategy `[cut,comp(7)]`, nanoCoP 1.0 combines the `[]` and the `[cut]` techniques. The "reo" version proves 242 new problems compared to nanoCoP 1.0, showing that reordering clauses and conjecture start clauses are effective techniques as well. Using a strategy scheduling, which is limited to the CPU time limit of 10 seconds (i.e., not all 20 strategies are used), nanoCoP 1.1 proves significantly more problems than nanoCoP 1.0.

Table 1. Evaluation of different optimization techniques

	"basic"	`[]`	`[conj]`	`[scut]`	`[cut]`	1.0	"reo"	1.1
proved	1465	1516	1682	1598	1691	1854	1855	**2138**
0 to 1sec.	1254	1294	1419	1350	1419	1620	1510	**1620**
1 to 10sec.	211	222	263	248	272	234	345	**518**
new proved	–	64	253	248	421	446	242	**335**
compared to	–	"basic"	`[]`	`[]`	`[]`	`[]`	1.0	1.0
refuted	131	133	–	–	–	133	–	**133**
total	1596	1649	1682	1598	1691	1987	1855	**2271**
errors	110	101	94	89	90	140	86	**11**

Table 2. Evaluation of different provers

	leanTAP	leanCoP	Prover9	E	nanoCoP 1.0	nanoCoP 1.1
proved	555	2541	2694	4125	2094	**2490**
0 to 1sec.	520	1643	1936	2783	1620	**1620**
1 to 10sec.	20	369	471	717	234	**234**
10 to 100sec.	15	529	287	625	240	**636**
refuted	0	67	0	479	134	**133**
total	555	2608	2694	4604	2228	**2623**
errors	1850	64	1456	446	448	**177**

5.2 Comparison with Other Provers

The performance of nanoCoP 1.1 is compared to the following provers: the lean (non-clausal) tableau prover leanTAP 2.3 [1], the resolution prover Prover9 (2009-02A) [6], the superposition prover E 1.9 [15] (using options "`--auto --tptp3-format`"), and leanCoP 2.2 (casc16) [7]. For leanTAP, leanCoP, and nanoCoP, the required equality axioms were added in a preprocessing step (which is included in the timings).

Table 2 shows the results of the evaluation for a CPU time limit of 100 seconds (the strategy scheduling of leanCoP 2.2 and nanoCoP 1.1 are adapted to this limit). The entry "errors" shows the number of problems for which the prover terminated without result before the CPU time limit was exceeded, e.g., because of a stack overflow (leanTAP, leanCoP, and nanoCoP) or an empty set-of-support (Prover9). nanoCoP 1.1 proves slightly less problems than Prover9 and about the same number of problems as leanCoP. The advantage of dealing directly with a non-clausal form is compensated by the overhead for dealing with the non-clausal data structure. Only few problems in the TPTP library are in a (deeply) nested non-clausal form; in this case proofs found by nanoCoP can be significantly shorter than the clausal proofs found by leanCoP [9]. nanoCoP 1.1 proves 150, 755, 301, and 452 problems not proved by leanCoP, Prover9, E, and nanoCoP 1.0, respectively.

6 Conclusion

The paper presented a few effective optimization techniques for non-clausal connection calculi and described their integration into the nanoCoP prover. Together with a strategy scheduling, these techniques are the basis for the nanoCoP 1.1 prover. It solves almost 400 more problems from the TPTP library than the first version of nanoCoP, which essentially consists only of the Prolog core prover.

Future work include an improved representation of the clauses stored in Prolog's database and the integration of further techniques to prune the search space, such as learning [4]. Furthermore, most of the presented techniques can be used for reasoning in non-classical, e.g., intuitionistic or modal first-order connection calculi [11, 13] as well, by taking the prefixes of the literals into account.

Acknowledgements. The author would like to thank Michael Färber for many useful discussions on the nanoCoP prover.

References

1. Beckert, B., Posegga, J.: leanTAP: lean, tableau-based deduction. Journal of Automated Reasoning 15(3), 339–358 (1995)
2. Gentzen, G.: Untersuchungen über das logische Schließen. Mathematische Zeitschrift 39, 176–210, 405–431 (1935)
3. Hähnle, R.: Tableaux and Related Methods. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 100–178. Elsevier, Amsterdam (2001)
4. 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)
5. Letz, R., Stenz, G.: Model Elimination and Connection Tableau Procedures. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 2015–2114. Elsevier, Amsterdam (2001)
6. McCune, W.: Release of Prover9. Mile high conference on quasigroups, loops and nonassociative systems, Denver (2005)
7. Otten, J.: Restricting backtracking in connection calculi. AI Communications 23, 159–182 (2010)
8. Otten, J.: A Non-clausal Connection Calculus. In: Brünnler, K., Metcalfe, G. (eds.) TABLEAUX 2011, LNAI, vol. 6793, pp. 226–241. Springer, Heidelberg (2011)
9. Otten, J.: nanoCoP: A Non-clausal Connection Prover. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016, LNAI, vol. 9706, pp 269–276. Springer, Heidelberg (2016)
10. Otten, J.: nanoCoP: Natural Non-clausal Theorem Proving. In Sierra, C. (ed.) IJCAI 2017, Sister Conference Best Paper Track, pp. 4924–4928. ijcai.org (2017)
11. Otten, J.: Non-clausal Connection Calculi for Non-classical Logics. In: Schmidt, R., Nalon, C. (eds.) TABLEAUX 2017, LNAI, vol. 10501, pp 209–227. Springer, Heidelberg (2017)
12. Otten, J., Bibel, W.: leanCoP: lean connection-based theorem proving. Journal of Symbolic Computation 36, 139–161 (2003)
13. Otten, J., Bibel, W.: Advances in Connection-Based Automated Theorem Proving. In: Hinchey, M. at al. (eds.) Provably Correct Systems. NASA Monographs in Systems and Software Engineering, pp. 211-241. Springer, Heidelberg (2017)
14. Reis, G.: Importing SMT and connection proofs as expansion trees. In: Kaliszyk, C., Paskevich, A. (eds.) 4th Workshop PxTP, EPTCS 186, pp. 3–10 (2015)
15. Schulz, S.: E – a brainiac theorem prover. AI Communications 15(2), 111–126 (2002)
16. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. Journal of Automated Reasoning 59(4), 483–502 (2017)