AI Communications 23 (2010) 159–182
DOI 10.3233/AIC-2010-0464
IOS Press

Restricting backtracking in connection calculi

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

Abstract. Connection calculi benefit from a goal-oriented proof search, but are in general not proof confluent. A substantial amount of backtracking is required, which significantly affects the time complexity of the proof search. This paper presents a simple strategy for effectively restricting backtracking in connection calculi. In combination with a few basic techniques it provides the basis for a refined connection calculus. The paper also describes how this calculus can be implemented directly by a few lines of Prolog code. This very compact program is the core of an enhanced version of the automated theorem prover leanCoP. The performance of leanCoP is compared with other lean theorem provers, connection provers, and state-of-the-art theorem provers. The results show that *restricted backtracking* is a successful technique when performing proof search in connection calculi.

Keywords: Automated theorem proving, connection calculus, restricted backtracking, leanCoP

1. Introduction

Connection calculi are a well-known basis to automate formal reasoning in classical first-order logic. Among these calculi are Bibel's connection method [3,4], the connection tableau calculus [18] and the model elimination calculus [19]. Their main inference step connects an atomic formula of the conjecture, or an atomic formula of the proof derivation, to a new atomic formula with the same predicate symbol but different polarity. The two connected atomic formulae are called a *connection*, which corresponds to a closed branch in the tableau framework [10] or an axiom in the sequent calculus [9]. The concept of a connection permits a goal-oriented proof search. While the goal-oriented strategy reduces the search space – compared to, e.g., standard tableau or sequent calculi – it is not confluent, i.e. it might end up in dead ends. To achieve completeness an extensive use of *backtracking* is required. There have been only a few attempts to limit this backtracking, e.g., by using a confluent connection calculus [1,4]. But the practical benefit of a confluent proof search does not outweigh the disadvantages introduced by limiting the goal-oriented proof search.

Another major problem in connection calculi is the integration of equality. Paramodulation, a successful technique for dealing with equality in saturation-based theorem proving, is not complete for the goal-oriented approach of connection calculi. Therefore equality is usually integrated by adding the axioms of equality, i.e. the axioms for reflexivity, symmetry, transitivity and substitutivity. In some cases several hundreds of equality axioms need to be added to the given formula.

This paper presents a simple strategy for restricting backtracking in connection calculi that significantly reduces the search space. The main idea is that once a literal has been solved, no alternative connections are considered anymore. This is achieved by cutting off any so-called *non-essential backtracking* that occurs after a literal is solved. Even though this strategy is incomplete, it performs very well in practice, in particular for problems that have many axioms or that contain equality axioms. Experimental results show that it is – up to now – the single most effective technique for pruning the search space in connection calculi.

Many different techniques have been proposed so far for pruning the search space in connection calculi; see, e.g., [4,18]. In this paper a set of basic techniques is selected that are most successful in practice. Among these are well-known techniques, such as regularity and lemmata. Together with the new technique for restricting backtracking, these pruning techniques are the main enhancements of the basic connection calculus.

This calculus can be implemented by a few lines of Prolog code. The resulting implementation is the core of leanCoP 2.0, a refined version of the theorem prover leanCoP [28]. A definitional transformation for translating first-order formulae into clausal form is presented as well. It is shown that this transformation is more appropriate for connection calculi than other established clausal-form transformations.

Outline of the paper

The paper is organized as follows. First some fundamental concepts are defined in Section 2. The basic connection calculus together with a few essential and well-known techniques for pruning the search space are presented in Section 3. It includes an original formalization of these techniques within the "sequent-style" connection calculus, as well as a new optimized definitional transformation into clausal form. Section 4 provides a comprehensive analysis of the amount of backtracking required in order to find a proof in the connection calculus. The new technique for restricting this backtracking is introduced afterwards. In Section 5 the basic calculus is specified in Prolog before the presented techniques for pruning the search space are added, leading to the leanCoP 2.0 core prover. Section 6 provides comprehensive experimental results of leanCoP on the problems in the TPTP library. Some improvements and extensions of the leanCoP implementation, e.g. to intuitionistic logic, are described in Section 7. The paper concludes with a short summary and a brief outlook on further research in Section 8.

2. Preliminaries

The reader is assumed to be familiar with the language of classical first-order logic, see, e.g., [8]. In this paper the letters P, Q, R, S, T are used to denote predicate symbols, a, b, c, d, e to denote constants and x, y, z to denote variables. Terms are denoted by s, t and are built from functions, constants and variables. *Atomic formulae* or *atoms* are built from predicate symbols and terms. The connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ denote negation, conjunction, disjunction and implication, respectively. A *(first-order) formula*, denoted by F, A, B, D, consists of atomic formulae, the connectives and the existential and universal quantifiers, denoted by $\exists$ and $\forall$, respectively. A *literal*, denoted by L, is either an atomic formula or a negated atomic formula. The complement $\overline{L}$ of a literal L is P if L is of the form $\neg P$, and $\neg L$ otherwise.

In the following, formulae are considered that are either in Skolemized negation normal form or in clausal form. A formula is in *negation normal form* if it contains only disjunctions, conjunctions and literals. A *clause*, denoted by C, is of the form $L_1 \wedge \cdots \wedge L_n$ where L_i is a literal. A formula in *disjunctive normal form* or *clausal form* has the form $C_1 \vee \cdots \vee C_n$ where C_i is a clause. A clause is often written as a set of literals $\{L_1, \ldots, L_n\}$. A formula in clausal form can also be written as a set of clauses $\{C_1, \ldots, C_n\}$ and is called a *matrix*, denoted by M. In the graphical representation of a matrix, its clauses are arranged horizontally, while the literals of each clause are arranged vertically. A *positive representation* is used throughout the paper, i.e. the introduced calculi are used to characterize *validity* and not unsatisfiability.[1]

Example 1 (First-order formula, clause, matrix).

$$\big(((\exists x Q(x) \vee \neg Q(c)) \Rightarrow P) \\ \wedge (P \Rightarrow (\exists y Q(y) \wedge R))\big) \Rightarrow (P \wedge R)$$

is a formula. Its equivalent negation normal form (where y is replaced by the Skolem term b) is

$$\big((Q(x) \vee \neg Q(c)) \wedge \neg P\big) \vee \big(P \wedge (\neg Q(b) \vee \neg R)\big) \\ \vee (P \wedge R)$$

and its equivalent clausal form is

$$(P \wedge R) \vee (\neg P \wedge Q(x)) \vee (\neg Q(b) \wedge P) \\ \vee (\neg Q(c) \wedge \neg P) \vee (P \wedge \neg R).$$

The matrix of this formula is

$$\big\{\{P, R\}, \{\neg P, Qx\}, \{\neg Qb, P\}, \\ \{\neg Qc, \neg P\}, \{P, \neg R\}\big\},$$

where some parentheses are omitted for simplicity. It consists of five clauses and can be represented in a two-dimensional graphical way:

$$\begin{bmatrix} P & \neg P & \neg Qb & \neg Qc & P \\ R & Qx & P & \neg P & \neg R \end{bmatrix}.$$

Besides the concept of a connection, paths and term substitutions are defined in the following.

[1]The difference is marginal but becomes more important when non-classical logics, such as intuitionistic logic (see Section 7.2), are considered within the presented framework.

Definition 1 (Connection, path, term substitution).

(1) A *connection* is a set that contains two literals of the form $\{P(s_1,\ldots,s_n), \neg P(t_1,\ldots,t_n)\}$.
(2) A *path* through a matrix $M = \{C_1,\ldots,C_n\}$ is a set of literals that contains one literal from each clause $C_i \in M$, i.e. $\bigcup_{i=1}^{n}\{L'_i\}$ with $L'_i \in C_i$.
(3) A *first-order* or *term substitution* σ is a mapping from the set of variables to the set of terms. In $\sigma(L)$ all variables of the literal L are substituted according to their mapping in σ.

Example 2 (Connection, path, term substitution). Consider the formula in Example 1 and its matrix. Then $\{P, \neg P\}$, $\{R, \neg R\}$, $\{Qx, \neg Qb\}$ and $\{Qx, \neg Qc\}$ are connections. $\{P, \neg P, \neg Qb, \neg Qc, \neg R\}$ and $\{R, Qx, \neg Qb, \neg Qc, P\}$ are, e.g., paths through the matrix. $\sigma(x) = c$ is a term substitution.

These concepts are the basis of the connection calculus presented in the next section.

3. Proof search in the connection calculus

At first the basic connection calculus is described, before introducing a definitional clausal-form transformation and the rules for regularity and lemmata.

3.1. The basic calculus

The connection calculus uses a connection-driven search strategy. In each inference step a connection is identified along an active (sub-)path and only paths not containing the active path and this connection will be considered afterwards. See, e.g., [3,4,28] for details.

Definition 2 (Connection calculus). The axiom and rules of the *connection calculus* are given in Fig. 1. The words of the calculus are tuples "C, M, Path" where the clause C is the *open subgoal*, M is the matrix of the given formula, and the *active path Path* is a subset of a path through M. In the rules of the calculus C_1 and C_2 are clauses, σ is a term substitution, and $\{L_1, L_2\}$ is a connection with $\sigma(L_1) = \sigma(\overline{L_2})$. The rules of the calculus are applied in an analytic (i.e. bottom-up) way. The term substitution σ is applied to the whole derivation.

Theorem 1 (Correctness and completeness). *A first-order formula M in clausal form is valid in classical logic iff there is a connection proof for "$\varepsilon, M, \varepsilon$", i.e. a derivation for "$\varepsilon, M, \varepsilon$" in the connection calculus so that all leaves are axioms.*

Axiom (Ax) $\quad \overline{\{\}, M, \mathit{Path}}$

Start rule (St)

$$\frac{C_2, M, \{\}}{\varepsilon, M, \varepsilon} \text{ and } C_2 \text{ is copy of } C_1 \in M$$

Reduction rule (Red)

$$\frac{C, M, \mathit{Path} \cup \{L_2\}}{C \cup \{L_1\}, M, \mathit{Path} \cup \{L_2\}} \text{ with } \sigma(L_1) = \sigma(\overline{L_2})$$

Extension rule (Ext)

$$\frac{C_2 \backslash \{L_2\}, M, \mathit{Path} \cup \{L_1\} \quad C, M, \mathit{Path}}{C \cup \{L_1\}, M, \mathit{Path}}$$

$$\text{and } C_2 \text{ is copy of } C_1 \in M,\ L_2 \in C_2,$$

$$\sigma(L_1) = \sigma(\overline{L_2})$$

Fig. 1. The basic connection calculus.

A proof of this theorem can be found in [4,18]. Proof search in the connection calculus is carried out by first applying the start rule and then repeatedly applying the reduction or the extension rule. The latter rules identify a connection $\{P(s_1,\ldots,s_n), \neg P(t_1,\ldots,t_n)\}$ with $\sigma(s_i) = \sigma(t_i)$, for $1 \leqslant i \leqslant n$. In the sequent calculus [9] this connection corresponds to an axiom of the form $P(\sigma(t_1),\ldots,\sigma(t_n)) \vdash P(\sigma(s_1),\ldots,\sigma(s_n))$, whereas the active path *Path* corresponds to the literals in the current sequent. The term substitution σ is calculated by one of the well-known algorithms for term unification, see, e.g., [20].

Example 3 (Connection calculus). Let $M = \{\{P, R\}, \{\neg P, Qx\}, \{\neg Qb, P\}, \{\neg Qc, \neg P\}, \{\neg R, P\}\}$ be the matrix of the formula in Example 1. A derivation for M in the connection calculus with $\sigma(x') = \sigma(x'') = c$ is given in Fig. 2. Since all leaves are axioms it represents a connection proof and therefore the corresponding formula is valid.

The presented connection calculus is very similar to the *connection tableau calculus* [17,18]. In the connection calculus (the active) *Path* corresponds to the set of literals on the currently considered branch of the connection tableau and the literals of the open subgoal clause C correspond to the open leaf nodes of the currently considered tableau branch. A connection proof

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\},M,\{P,Qx'\}}\,Ax}{\{\neg P\},M,\{P,Qx'\}}\,Red \quad \dfrac{}{\{\},M,\{P\}}\,Ax}{\{Qx'\},M,\{P\}}\,Ext \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\},M,\{R,P,Qx''\}}\,Ax}{\{\neg P\},M,\{R,P,Qx''\}}\,Red \quad \dfrac{}{\{\},M,\{R,P\}}\,Ax}{\{Qx''\},M,\{R,P\}}\,Ext \quad \dfrac{}{\{\},M,\{R\}}\,Ax}{\{P\},M,\{R\}}\,Ext \quad \dfrac{}{\{\},M,\{\}}\,Ax}{\{R\},M,\{\}}\,Ext}{\dfrac{\{P,R\},M,\{\}}{\varepsilon,\{\{P,R\},\{\neg P,Qx\},\{\neg Qb,P\},\{\neg Qc,\neg P\},\{\neg R,P\}\},\varepsilon}\,St}\,Ext$$

Fig. 2. A connection proof in the connection calculus.

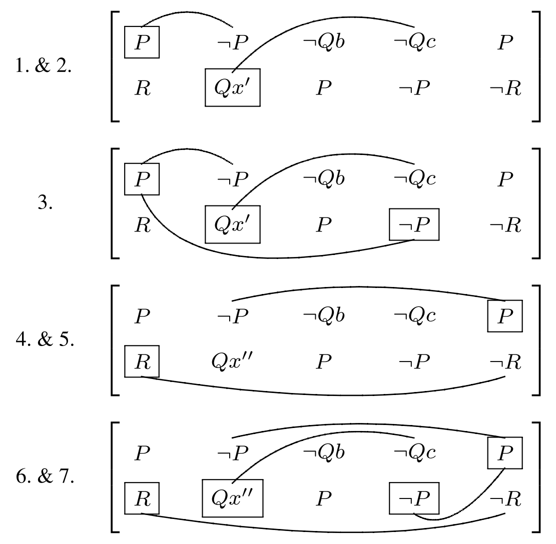


Fig. 3. A connection proof using the graphical matrix representation.

can also be represented by the graphical matrix presentation [3,4].

Example 4 (Graphical connection (tableau) proof). The connection proof in Fig. 2 from Example 3 can be illustrated by the graphical representation in Fig. 3. The literals of each connection of the connection proof in Fig. 2 are connected with a line. The literals of the active path are boxed. While the extension steps connect a literal to a new clause (steps 1, 2, 4–6), the reduction steps connect to literals in the active path (steps 3 and 7). Together with the substitution $\sigma(x') = \sigma(x'') = c$ these matrices represent a connection proof. The corresponding connection tableau proof is depicted in Fig. 4. Every connection corresponds to one tableau leaf and the literals of the active path correspond to the literals on the tableau branches.

The following matrix characterization [4] of classical validity can be seen as the underlying basis of the connection calculus. The notion of *multiplicity* is used

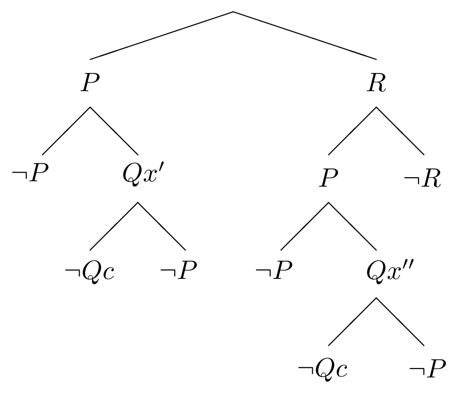


Fig. 4. A connection proof using the tableau representation.

to encode the number of clause copies used in a connection proof. It is a function $\mu : M \to \mathbb{N}$ that assigns each clause in a matrix M a natural number specifying how many copies of this clause are considered for the proof. The matrix that includes these copies is denoted by M^{μ}. Clause copies correspond to applications of the contraction rule in the sequent calculus [9].

Lemma 1 (Matrix characterization). *A matrix M is classically valid iff there exist a multiplicity μ, a term substitution σ and a set of connections $\mathcal{C}$, such that every path through M^{μ} contains a complementary connection $\{L_1, L_2\} \in \mathcal{C}$, i.e. a connection with $\sigma(L_1) = \sigma(\overline{L_2})$. The tuple $(\mu, \sigma, \mathcal{C})$ is called a matrix proof.*

It is important to notice that the matrix characterization is *not* a calculus, i.e. it does not provide any information on how to actually calculate the tuple $(\mu, \sigma, \mathcal{C})$. A thorough analysis of the relation between the matrix characterization and the connection (tableau) calculus is given in [15]. A connection proof can be seen as a matrix proof using an appropriate multiplicity μ. Because of the close relationship between these proof representations, the representation that is most appropriate for an explanation will be used throughout the rest of this paper. There are matrix characterizations for a wide set of logics, e.g. for intuitionistic, modal, and linear logic [13,45]. Section 7.2 gives more details for intuitionistic logic.

Example 5 (Matrix characterization). Consider the matrix M from Example 3 and its graphical representation

$$\begin{bmatrix} P & \neg P & \neg Qb & \neg Qc & P^* \\ R & Qx & P' & \neg P' & \neg R \end{bmatrix}$$

in which literals that occur more than once are marked to distinguish them from each other. Then the tuple $(\mu, \sigma, \mathcal{C})$ with $\mu(i) = 1$ for $i = 1, \ldots, 5$, $\sigma(x) = c$, and $\mathcal{C} = \{\{P, \neg P\}, \{Qx, \neg Qc\}, \{\neg P', P\}, \{R, \neg R\}, \{P^*, \neg P\}, \{\neg P', P^*\}\}$ is a matrix proof for M.

If a matrix has no *positive* clause, i.e. a clause with no negation, then there is a path that contains only negated atoms and the matrix cannot be valid. Therefore every connection proof has to contain a positive clause and the following proposition holds.

Proposition 1 (Positive start clause). *The connection calculus remains correct and complete if the clause C_1 of the start rule is restricted to positive clauses.*

3.2. Definitional clausal form

The presented connection calculus works on formulae in clausal form. Formulae that are not in this form have to be translated into clausal form. The *standard transformation* translates a first-order formula F that is in negation normal form into clausal form by applying the following distributivity rules to all subformulae of F until they cannot be applied anymore:

$$(A \vee B) \wedge D \equiv (A \wedge D) \vee (B \wedge D),$$
$$A \wedge (B \vee D) \equiv (A \wedge B) \vee (A \wedge D).$$

In the worst case the size of the formula F will grow exponentially and thus increase the search space for a proof in the connection calculus significantly. The *structure-preserving* or *definitional transformation* into clausal form [7,31] avoids this disadvantage by introducing definitions for all subformulae. Optimized versions of this translations reduce the number of clauses and terms by reducing the number of definitions [24]. Whereas this approach seems to improve performance for saturation-based calculi, practical evaluations have shown that this is not always the case for connection calculi (see Section 6.1). Therefore a different approach is used, where definitions are introduced only for subformulae of the form $A \vee B$ that occur within a conjunction, i.e. within a formula of the form $(A \vee B) \wedge D$ or $D \wedge (A \vee B)$.

Definition 3 (Definitional clausal form). Let F be a formula in negation normal form and let *cla*(D) be the standard transformation of a formula D into clausal form. The *definitional tuple* $(F', \mathcal{D})$ of F, where $\mathcal{D}$ is a set of formulae, is inductively defined as follows:

(1) If F is a literal, then $(F, \{\})$ is the definitional tuple of F; otherwise
(2) if F is of the form $A \vee B$ and F occurs within a conjunction and $(A', \mathcal{D}_A)$ and $(B', \mathcal{D}_B)$ are the definitional tuples of A and B, respectively, then $(S(x_1, \ldots, x_n), \{\neg S(x_1, \ldots, x_n) \wedge A', \neg S(x_1, \ldots, x_n) \wedge B'\} \cup \mathcal{D}_A \cup \mathcal{D}_B)$ is the definitional tuple of F, where S is a new predicate symbol and $x_1, \ldots, x_n$ are the variables occurring in $(A \vee B)$; otherwise
(3) F is of the form $A \circ B$ with $\circ \in \{\wedge, \vee\}$ and if $(A', \mathcal{D}_A)$ and $(B', \mathcal{D}_B)$ are the definitional tuples of A and B, respectively, then $(A' \circ B', \mathcal{D}_A \cup \mathcal{D}_B)$ is the definitional tuple of F.

Then the *definitional clausal form* of F is defined as $F' \vee$ *cla*$(D_1) \vee \cdots \vee$ *cla*(D_n) where $(F', \{D_1, \ldots, D_n\})$ is the definitional tuple of F.

Lemma 2 (Definitional clausal form). *A formula F is valid iff its definitional clausal form F' is valid.*

The proof is by structural induction on the size of the formula F. It is shown that all paths through the matrix M of F contain a complementary connection if, and only if, all paths through the matrix M' representing the definitional transformation F' of F contain a connection (see Fig. 5). As the proof is conducted in a purely proof-theoretical way, i.e., based on the matrix characterization of validity, it can be adapted to intuitionistic logic as well (see Section 7.2).

Example 6 (Definitional clausal form). Consider the formula $((Q(x) \vee \neg Q(c)) \wedge \neg P) \vee (P \wedge (\neg Q(b) \vee \neg R)) \vee (P \wedge R)$ in negation normal form from Example 1. The upper matrix in Fig. 6 shows its standard transformation into clausal form; the lower matrix represents its definitional clausal form. The definitional

2. $$M_1 = [\, A' \quad \mathcal{D}_A \quad B' \quad \mathcal{D}_B \,]$$

$$M_1' = \begin{bmatrix} & \neg S(x_1,\ldots,x_n) & \neg S(x_1,\ldots,x_n) & \mathcal{D}_A & \mathcal{D}_B \\ S(x_1,\ldots,x_n) & A' & B' & & \end{bmatrix}$$

3. a $$M_2 = \begin{bmatrix} [\, A' \; \mathcal{D}_A \,] \\ [\, B' \; \mathcal{D}_B \,] \end{bmatrix} \qquad M_2' = \begin{bmatrix} A' & \mathcal{D}_A & \mathcal{D}_B \\ B' & & \end{bmatrix}$$

3. b $$M_3 = [\, A' \quad \mathcal{D}_A \quad B' \quad \mathcal{D}_B \,] \qquad M_3' = [\, A' \quad B' \quad \mathcal{D}_A \quad \mathcal{D}_B \,]$$

Fig. 5. The definitional clausal-form transformation.

$$\begin{bmatrix} P & \neg P & \neg Qb & \neg Qc & P \\ R & Qx & P & \neg P & \neg R \end{bmatrix}$$

$$\begin{bmatrix} P & \neg P & \neg Sx & \neg Sx & P & \neg T & \neg T \\ R & Sx & Qx & \neg Qc & T & \neg Qb & \neg R \end{bmatrix}$$

Fig. 6. Standard and definitional clausal form.

translation introduces definitions for $(Q(x) \vee \neg Q(c))$ and $(\neg Q(b) \vee \neg R)$, which are named Sx and T, respectively (Sx and $\neg Sx$ can also be simplified to S and $\neg S$, respectively). The lower matrix consists of more clauses but allows fewer combination of connections. For example, there are only two P, each with only one choice to choose a connection, instead of three P, each with two choices for possible connections in the standard clausal form. Fewer connections reduce backtracking when searching for a connection proof.

3.3. Regularity and lemmata

Regularity and lemmata are well-known inference rules for pruning the search space in connection calculi. See, e.g., [17,18] for details.

Definition 4 (Regularity). A connection proof is *regular* iff no literal occurs more than once in the active path.

Since the active path corresponds to the set of literals in a branch in the connection tableau representation, a connection tableau proof is regular if in the currently considered branch no literal occurs more than once. For example, on the left side of Fig. 7 the literal L occurs twice in the tableau branch, hence the tableau is not regular. The regularity condition is integrated into

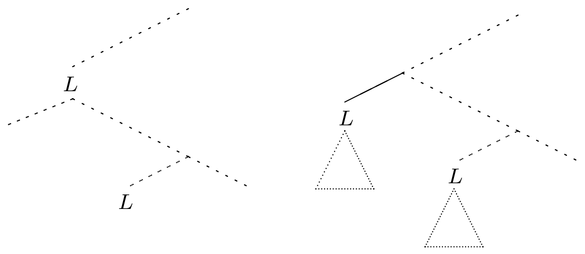


Fig. 7. Regularity and lemmata in the connection calculus.

the connection calculus in Fig. 1 by imposing the following restriction on the reduction and extension rule:

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

Lemma 3 (Regularity). *A formula M in clausal form is valid iff there is a regular connection proof for "$\varepsilon, M, \varepsilon$".*

Regularity is correct, since it only imposes a restriction on the applicability of the reduction and extension rules. The completeness proof can be found in [18]. Regularity is so far considered the most effective single technique to prune the search space in connection calculi [18]. Another important technique is the reuse of subproofs, named lemmata or factorization.

Definition 5 (Lemmata). The connection calculus in Fig. 1 is modified by adding a set of literals *Lem*, called *lemmata*, to all tuples "C, M, Path". The empty set $\{\}$ is added to the premise of the new start rule, ε is added to its conclusion. The set $\mathit{Lem} \cup \{L_1\}$ is added to the premise of the new reduction rule and the right premise of the extension rule. Furthermore, the following rule is added to the connection calculus:

Lemma rule

$$\frac{C, M, \mathit{Path}, \mathit{Lem} \cup \{L_2\}}{C \cup \{L_1\}, M, \mathit{Path}, \mathit{Lem} \cup \{L_2\}}$$
with $\sigma(L_1) = \sigma(L_2)$.

In the connection tableau calculus this technique is named *factorization* and uses an additional dependency relation on the tableau nodes [18].

Lemma 4 (Lemmata). *A formula M in clausal form is valid iff there is a (regular) connection proof for "$\varepsilon, M, \varepsilon, \varepsilon$" in the connection calculus with lemmata.*

The correctness and completeness of the connection calculus with lemmata (and regularity) follows immediately from the fact that subproofs can be reused as illustrated in the connection tableau on the right side of Fig. 7. See also [17,18] for details.

Example 7 (Regularity and lemmata). Consider the matrix from Example 3 and its proof shown in Fig. 3. The matrix is depicted in the upper part of Fig. 8. If for the second extension step the third clause $\{\neg Qb, P\}$ is selected, the regularity condition is violated since P already occurs in the active path $\{P, Qx\}$. Consider the lower matrix in Fig. 8, which shows the connection proof after three proof steps. When proving the literal R the literal P (boxed twice) is a lemma. After the extension step to the clause $\{P, \neg R\}$ the connection proof is completed by applying the lemma rule for the literal P. Consider the right branch of the connection proof in Fig. 2. After the extension step to the clause $\{P, \neg R\}$ the lemma rule can be applied to $\{P\}, M, \{R\}, \{P\}$. Afterwards the whole branch can be immediately closed by an axiom.

In this section the basic connection calculus and a few additional techniques and inference rules to prune the search space have been introduced: positive start clauses, definitional clausal form, regularity, and lemmata. The next section introduces a new technique for pruning the search space in connection calculi.

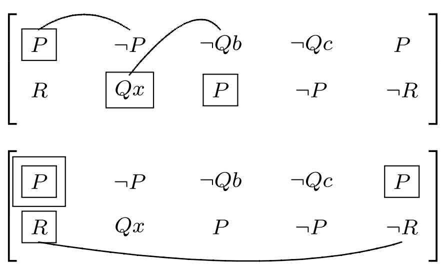


Fig. 8. Using regularity and lemmata in a connection proof.

4. Restricting backtracking in connection calculi

In contrast to saturation-based calculi, such as resolution [34] and instance based methods [14], connection calculi are not proof confluent. A significant amount of backtracking is required during the proof search. In this section it is first clarified for which rules backtracking might be required when searching for a connection proof. Afterwards a comprehensive analysis of the amount of backtracking actually used to find connection proofs is given before an approach for restricting this backtracking is introduced.

4.1. Proof search and backtracking

In general backtracking is used if a calculus has more than one rule that can be applied to a node in a derivation. In this case the search algorithm first chooses the first applicable rule. If the application of this rule does not lead to a proof the next applicable rule is chosen and so on.

Proposition 2 (Backtracking in connection calculi). *For the proof search in the connection calculus shown in Fig.* 1 *backtracking is required:*

(1) *for different (positive) start clauses C_1 of the start rule,*
(2) *for different literals L_2 of the reduction rule,*
(3) *for different clauses C_1 and different literals L_2 of the extension rule,*
(4) *for different literals L_2 of the lemma rule, and*
(5) *if more than one of the reduction, extension, or lemma rules are applicable at the same time.*

No backtracking is required when choosing the literal L_1 in the reduction or the extension rule, since all literals in $C \cup \{L_1\}$ will be considered in subsequent proof steps anyway.

Since the term substitution σ is *rigid* for the entire derivation, it is not only important *that* a branch of the derivation is closed, but *how* it is closed. The application of different rules to a node might result in different substitutions. In order to consider alternative substitutions, backtracking has to be carried out even for branches of a derivation that have already been closed.

Example 8 (Backtracking in connection calculi). Consider the matrix $\{\{Pa\}, \{\neg Px, Pb\}, \{\neg Py, \neg Pz, Qz\}, \{Pc, Pd\}, \{Pe\}, \{Qe\}\}$ in Fig. 9. The derivation of a proof is started with the clause $\{Pa\}$. The first extension step connects to the literal $\neg Px$ with $\sigma(x) = a$,

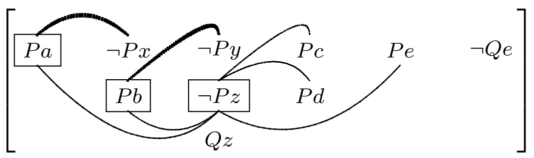


Fig. 9. Backtracking in the connection calculus.

the second one connects to the literal $\neg Py$ with $\sigma(y) = b$; in the matrix representation these steps are marked by thick lines. From the literal $\neg Pz$ there are five possible connections, which are marked by thin lines: to literals of the active path $\{Pa, Pb\}$ by applying the reduction rule or to literals of the fourth clause $\{Pc, Pd\}$ or the fifth clause $\{Pe\}$ by applying the extension rule. Depending on which of these literals is chosen the proof results in one of the substitutions $\sigma(z) = x$ with $x \in \{a, b, c, d, e\}$. But only the substitution $\sigma(z) = e$ ensures that the proof can be completed with the connection $\{Qz, \neg Qe\}$.

4.2. *Analysing backtracking in connection proofs*

When searching for a connection proof the aim is to eliminate literals from an open subgoal (clause). Each literal of an open subgoal corresponds to an open branch in the connection tableau representation. The notion of a *solved* literal is used to express the fact that within a derivation a proof step deletes the so-called *principal* literal from an open subgoal and any new open subgoal introduced by this proof step can be solved as well.

Definition 6 (Principal literal, solved literal). When the reduction, extension or lemma rules are applied the literal L_1 (see Fig. 1 and Definition 5) is called the *principal literal* of the proof step. A reduction or lemma step *solves a literal* L iff L is the principal literal of the proof step. An extension step *solves a literal* L iff L is the principal literal of the proof step and there is a proof for the left premise, i.e. there is a derivation for the left premise so that all leaves are axioms.

A solved literal in the connection calculus corresponds to a closed branch in the tableau representation.

Example 9 (Principal literal, solved literal). Consider the matrix of Example 8 in Fig. 9. The principal literal of the third extension step is $\neg Pz$. The reduction steps to Pa and Pb, as well as the extension step to Pe solves the literal $\neg Pz$. The extensions steps to Pc or Pd solve $\neg Pz$ as well after Pd or Pc, respectively, are solved using a copy of the third clause.

In the following, the amount of backtracking required for finding connection proofs is evaluated. For this purpose all non-clausal so-called FOF problems of version 3.7.0 of the TPTP problem library [40] are considered (see also remarks in Section 6). To find connection proofs the "regular" variant of the leanCoP 2.0 core prover is used that implements the basic calculus with regularity, lemmata, and the presented definitional clausal form. Details of the implementation are given in Sections 5 and 6.2. During the proof search the lemma rule is applied before the reduction rule, which is applied before the extension rule; the left premise of the extension rule is considered first.

At first the formula AGT016+2 is considered. It is included in the AGT domain, which contains problems that formalize reasoning about agents. The clausal form of this problem has more than 1000 clauses including the equality axioms. The connection proof found by leanCoP consists of one start step and nine extension steps. These ten proof steps are shown in Table 1. The application of the axiom is deterministic, i.e., whenever the axiom rule is applicable, no other rule can be applied. For that reason the axiom rule is not considered in the table and the following analysis.

The third, fourth and fifth column of Table 1 show for each proof step with principal literal L the total number of applicable rules with the same principal literal L, the rule number that has first solved the literal L, and the rule number that is used in the actual connection proof. For the start step the third column shows the number of applicable start rules, i.e. the first line indicates that there are 48 positive start clauses, of which the first one is used in the proof. The second line, e.g., indicates that the second proof step is an extension step; there are alltogether six applicable rules with the same principal literal, the fourth applicable rule is

Table 1

Backtracking in the connection proof for AGT016+2

Proof step	Applied rule	Applicable rules	Solved rule #	Proof rule #
1	start	48	–	1
2	extension	6	4	4
3	extension	5	5	5
4	extension	7	1	1
5	extension	6	1	1
6	extension	10	1	1
7	extension	11	1	1
8	extension	10	1	1
9	extension	6	5	5
10	extension	4	4	4

the first one that solves this literal, and the fourth applicable rule is also the one used in the final connection proof. For six of the ten poof steps the first applicable rule is also the one used in the connection proof. More remarkable, for all ten proof steps the first rule that solves a literal is also the one used in the proof. To see if this property holds for other connection proofs as well, all 17 problems in the AGT domain for which a connection proof is found are now considered.

Table 2 shows a summary of all 98 proof steps used in the 17 connection proofs for the formulae of the AGT domain. The first section shows the statistics for the start step. For example, the first line indicates that there are five problems (fourth column) each with 43 possible start clauses (first column) and in each case the first start clause is used in the proof (third column). The second section contains the statistics about the reduction and extension steps. For example, the first line shows that there are 10 proof steps (fourth column) for which there are four applicable rules with the same principal literal (first column), the fourth applicable rule is the first one that solves this literal (second column), and the fourth applicable rule is the one used in the connection proof. There are two observations:

(1) Even though there are between 43 and 49 alternatives for choosing a start clause, in 15 of the 17 proofs the selection of the first start clause results in a successful proof search.
(2) For 79 of the 81 reduction or extension steps, the first applicable rule that solves a literal is also the one used in the proof. Only for two proof steps (sixth line and marked with a "*") this property does not hold; in these two cases there are six applicable rules, the fourth applicable rule is the first one that solves the literal, and the fifth applicable rule is the one used in the connection proof.

Table 2

Backtracking in connection proofs for the AGT domain

Applicable rules	Solved rule #	Proof rule #	Number of occurrences
		Start rule	
43	–	1	5
43	–	36	1
44	–	1	2
48	–	1	5
48	–	36	1
49	–	1	3
		Reduction and extension rule	
4	4	4	10
5	3	3	5
5	5	5	12
6	1	1	12
6	4	4	8
6	4	5	*2
6	5	5	5
6	6	6	11
7	1	1	4
9	1	1	4
10	1	1	6
11	1	1	2

These findings suggest a distinction between backtracking that occurs before a literal is first solved and backtracking that occurs afterwards. The notion of *essential backtracking* is used for the former kind of backtracking. Proof steps that involve only essential backtracking are so-called *essential proof steps*.

Definition 7 (Essential backtracking/proof step). Let $R_1, \ldots, R_n$ be instances of rules with the same principal literal L_1 applicable to a node of a derivation in the connection calculus. If the literal L_1 can be solved by applying the rule R_i, but not by applying the rules $R_1, \ldots, R_{i-1}$, then backtracking over the rules $R_2, \ldots, R_i$ is called *essential backtracking*; backtracking over the rules $R_{i+1}, \ldots, R_n$ is called *non-essential backtracking*. The application of one of the rules $R_1, \ldots, R_i$ is an *essential proof step*; the application of one of the rules $R_{i+1}, \ldots, R_n$ is a *non-essential proof step*.

Whereas essential backtracking is necessary to close a branch in the connection calculus, non-essential backtracking might additionally be required in order to find alternative term substitutions.

Example 10 (Essential backtracking/proof step). Consider the matrix in Example 8 after two extension steps. There are five applicable rules with the principal literal $\neg Pz$, namely connections to Pa, Pb, Pc, Pd and Pe. Since the connection to Pa already solves the literal $\neg Pz$, backtracking over the other connections/rules is non-essential backtracking. Only the connection to Pa is an essential proof step.

Of the 17 proofs for the problems of the AGT domain, 15 proofs contain only essential proof steps. Although non-essential backtracking does happen during the *search* for these proofs, the proofs itself could be found using only essential backtracking. Only two proofs involve non-essential proof steps and can only be found using non-essential backtracking as well.

To conclude the analysis all problem domains of the TPTP library are now considered. The "regular" variant of leanCoP proves 1256 out of 5051 problems. The statistics for these problems are given in Table 3. For each domain the following information is given: number of proved problems (second column), number of proofs that do not use backtracking for the start step (third column), number of essential/non-essential proof steps of the proofs (fourth/fifth column), and number of proofs that contain only/not only essential proof steps (sixth/seventh column). The number of (non-)essential proof steps include applications of the lemma rule; the number of essential proof steps include the start step. The 1256 proofs consist of 21,888 proof steps, resulting in an average of about 17 proof steps per proof. Of the 1256 problems 981 (78%) are proved using the first start clause. 19,403 (89%) of these steps are essential proof steps. 882 (70%) of the 1256 proofs contain only essential proof steps.

Remarkable is the large number of essential proof steps and the large number of proofs that contain only essential proof steps. In the SWC domain even all 54 proof steps are essential and therefore the 14 proofs contain only essential proof steps. Although for the proof steps itself only essential backtracking is carried out, in general a significant amount of non-essential backtracking occurs during the actual proof search. This suggests to restrict backtracking during the proof search in a way that only allows essential backtracking.

Table 3
Backtracking in connection proofs for the TPTP problems

Domain	# of proofs	1st start clause	Essent. steps	Non-es. steps	Essent. proofs	Non-es. proofs
AGT	17	15	96	2	15	2
ALG	33	30	4489	1505	13	20
CAT	1	1	12	2	0	1
COM	1	1	127	6	0	1
CSR	93	87	605	57	75	18
GEO	153	84	1594	84	102	51
GRA	4	3	39	3	3	1
GRP	3	2	109	11	0	3
HAL	1	1	14	4	0	1
KRS	92	44	2752	144	60	32
LAT	1	0	26	1	0	1
LCL	26	26	219	63	8	18
MGT	35	30	877	66	15	20
MCS	2	2	40	6	1	1
NLP	8	8	810	6	5	3
NUM	36	32	254	16	25	11
PUZ	6	5	113	17	3	3
SET	193	141	2005	227	117	76
SEU	167	142	1997	167	99	68
SWC	14	14	54	0	14	0
SWV	160	117	1297	51	135	25
SYN	204	190	1734	41	189	15
TOP	6	6	139	6	3	3
total	1256	981	19,403	2485	882	374
[%]	100%	**78%**	**89%**	11%	**70%**	30%

4.3. Restricted backtracking

The main idea for restricting backtracking is to avoid backtracking once a literal has been solved. This is achieved by allowing only essential backtracking for reduction, extension and lemma steps. Furthermore, the start step can be restricted to the first start clause.

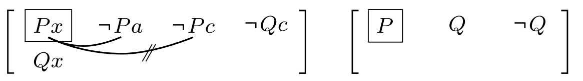
Definition 8 (Restricted backtracking/start step).

(1) Let $R_1, \ldots, R_i, \ldots, R_n$ be the instances of (reduction, extension or lemma) rules with principal literal L_1 that are applicable to a node of a derivation in the connection calculus and rule R_i solves L_1. *Restricted backtracking* does not apply the alternative rules $R_{i+1}, \ldots, R_n$ anymore.
(2) Let $C'_1, \ldots, C'_n$ be the possible start clause C_1 for the start rule. The *restricted start step* does not consider the alternative start clauses $C'_2, \ldots, C'_n$ anymore.

Restricted backtracking cuts off non-essential backtracking, while the restricted start step cuts off any alternative start clause. Restricted backtracking and the restricted start step preserve correctness of the connection calculus, but completeness is lost in either case.

Lemma 5 (Restricted backtracking/start step). *A formula M is valid if the proof search for M in the connection calculus using restricted backtracking and/or the restricted start steps succeeds. These search strategies are incomplete in the sense that for some valid formulae the proof search using restricted backtracking or the restricted start step does not succeed.*

Restricted backtracking and the restricted start step preserve correctness as the proof search space is only pruned. Completeness though is lost as the valid formulae $(Px \wedge Qx) \vee \neg Pa \vee \neg Pc \vee \neg Qc$ and $P \vee Q \vee \neg Q$ presented by the following matrices show:



After the first step using the connection $\{Px, \neg Pa\}$ with $\sigma(x) = a$ solves Px in the left matrix, the connection $\{Px, \neg Pc\}$, required for a proof, is not considered anymore. In the right matrix the restricted start step prevents the use of the alternative start clause $\{Q\}$.

Example 11 (Restricted backtracking). Consider the matrix in Fig. 10. After two extensions steps using the connections $\{Pa, \neg Px\}$ and $\{Qy, \neg Qb\}$ with $\sigma(x) = a$ and $\sigma(y) = b$, and a reduction step using the connection $\{\neg Pa, Pa\}$, the literal Ry cannot be solved anymore. Backtracking will not consider the second connection $\{Qy, \neg Qc\}$ anymore as Qy has already been solved. But the alternative connection $\{Pa, \neg Pa\}$ for the first extension step will still be considered as the literal Pa was not solved so far.

To restrict backtracking in this way seems to be too strict. But in Section 6 it is shown that this is not the case. In fact the approach turns out to be very successful in practice as the amount of backtracking is reduced significantly. For example, for problem AGT016+2 the "regular" proof takes 84 s and the proof search requires 312,831 inference steps. In contrast, the "restrict" variant of leanCoP (see Section 6.2) using restricted backtracking needs less than 0.3 s using only 427 inference steps for the proof search.

It is important to notice that a successful proof search using restricted backtracking is not limited to problems whose "regular" proof contains only essential proof steps, e.g. it is not limited to the 882 TPTP problems listed in Table 3. Proof search with restricted backtracking is able to solve problems whose "regular" proof contains non-essential proof steps as well. These proofs might have different lengths and/or use different connections. For example, of the 374 proofs that contain non-essential proof steps (see Table 3), 218 problems can be solved with restricted backtracking as well (within a time limit of 600 s). In addition 330 new problems are solved for which the "regular" variant of leanCoP was not able to find a proof. Section 6 provides more details about the performance of restricted backtracking.

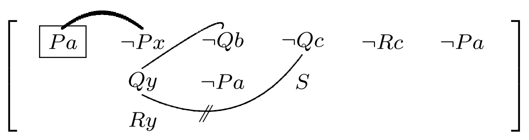


Fig. 10. Restricted backtracking.

5. An implementation

In this Section it is shown how the refined connection calculus presented in Section 3 including restricted backtracking from Section 4 can be specified by a few lines of Prolog code. The resulting Prolog implementation is the core of the leanCoP 2.0 theorem prover. The program is developed step by step. The description starts with the basic connection calculus and adds the additional techniques afterwards. See, e.g., [5] for an introduction to Prolog.

5.1. The basic calculus

The implementation of the basic connection calculus presented in Section 3.1 is shown in Fig. 11. A derivation for a formula in clausal form is generated by first applying the start rule and then repeatedly applying the reduction or the extension rule. Open branches are selected in a depth-first way.

The tuple $C, M, \mathit{Path}, \mathit{Lem}$ in the connection calculus is represented by the Prolog lists `Cla`, `Path` and `Lem`, which represent the open subgoal C, the active path *Path*, and the set of lemmata *Lem*, respectively. The matrix M is written into Prolog's database before the actual proof search starts. For every clause $C \in M$ and for every literal `L` $\in C$ the fact `lit(L,Cl,Grnd)` is stored, where `Cl` $= C \backslash \{$`L`$\}$ and `Grnd` is `g` if C is ground, otherwise `Grnd` is `n` (see below for an example). Atoms are represented by Prolog atoms, negation by "`-`". The substitution σ is stored implicitly by Prolog. The predicate

```
prove(Cla,Path,PathLim,Lem,Set)
```

implements the axiom, the reduction rule and the extension rule of the basic connection calculus of Fig. 1. This predicate succeeds (using iterative deepening as explained below) if, and only if, there is a connection proof for the tuple represented by the lists `Cla`, `Path`, `Lem`, and the matrix stored in Prolog's database represented by the `lit` predicate with $|$`Path`$| <$ `PathLim` where `PathLim` is the maximum size of the active `Path`. The *setting* `Set` is a list of options used to control the proof search and is explained in Section 5.5.

```
 (1)   prove([],_,_,_,_).

 (2)   prove([Lit|Cla],Path,PathLim,Lem,Set) :-
 (3)                                      % regularity
 (4)       (-NegLit=Lit;-Lit=NegLit) ->
 (5)          (                           % lemmata
 (6)                                      %
 (7)            member(NegL,Path), unify_with_occurs_check(NegL,NegLit)
 (8)            ;
 (9)            lit(NegLit,Cla1,Grnd1),
(10)                                      % iterative deepening
(11)                                      %
(12)            prove(Cla1,[Lit|Path],PathLim,Lem,Set)
(13)          ),
(14)                                      % restricted backtracking
(15)          prove(Cla,Path,PathLim,Lem,Set).
```

Fig. 11. The implementation of the basic connection calculus.

Line 1 implements the axiom, line 4 calculates the complement of the first literal `Lit` in `Cla`, which is used as the principal literal for the next reduction or extension step. The reduction rule is implemented in lines 7 and 15. In line 7 it is checked whether the active path `Path` contains a literal `NegL` that unifies with the complement `NegLit` of the principal literal `Lit`. In this case the alternative lines after the semicolon are skipped and the proof search for the premise of the reduction rule is invoked in line 15. The extension rule is implemented in lines 9, 12 and 15. In line 9 the predicate `lit(NegLit,Cla1,Grnd1)` is used to find a clause that contains the complement `NegLit` of the principal literal `Lit`.[2] `Cla1` is the remaining set of literals of the selected clause and the new open subgoal of the left premise. The proof search for the left premise of the extension rule, in which the active path `Path` is extended by the principal literal `Lit`, is invoked in line 12. Afterwards the proof search for the right premise is invoked in line 15. The lines implementing regularity, lemmata, iterative deepening, and restricted backtracking are added afterwards.

The start rule of the connection calculus is implemented as follows:

```
(a) prove(PathLim,Set) :-
(b)      prove([-(#)],[],PathLim,[],Set).
(c)                        % restricted start step
```

When the matrix M is written into Prolog's database the special literal # is added to all positive clauses. The proof search is then started with the open subgoal `[-#]` and an empty active path `[]`. Thus by default all positive clauses are used as possible start clauses. The predicate

```
prove(PathLim,Set)
```

succeeds if, and only if, there is a connection proof for the set of clauses stored in the database, for which the size of the active path is smaller than `PathLim`. Again `Set` is a list of search options described in Section 5.5.

Lean Prolog technology

The code in Fig. 11 is similar to the leanCoP 1.0 code [28]. In leanCoP 1.0 the matrix M is added as an argument to the `prove` predicates. For the extension step all clauses of the matrix M and all literals of each clause are searched for a suitable literal `NegLit`.[3] In leanCoP 2.0 the clauses are stored in Prolog's database and the goal `lit(NegLit,Cla1,Grnd1)` is used to find appropriate literals `NegLit`. This technique utilizes Prolog's built-in indexing mechanism on the first argument to quickly find connections. It integrates the main advantage of the "Prolog technology theorem proving" approach [37,38] into the lean theorem proving framework and improves performance (see Section 6.2).

Example 12 (Lean Prolog technology). Consider the matrix $\{\{P, R\}, \{\neg P, Qx\}, \{\neg Qb, P\}, \{\neg Qc, \neg P\},$

[2]Sound term unification has to be used when this predicate is called. In ECLiPSe Prolog sound unification is switched on with `set_flag(occur_check,on)`.

[3]For a matrix `M` this is can be achieved by using the following Prolog code: `append(MA,[C1|MB],M), copy_term(C1,C2), append(CA,[NegLit|CB],C2)`.

$\{P, \neg R\}\}$ of Example 1. It is stored in Prolog's database in the following form:

```
lit(#,[p,r],g).
lit(p,[#,r],g).         lit(r,[#,p],g).
lit(-p,[q(X)],n).       lit(q(X),[-p],n).
lit(-q(b),[p],g).       lit(p,[-q(b)],g).
lit(-q(c),[-p],g).      lit(-p,[-q(c)],g).
lit(p,[-r],g).          lit(-r,[p],g).
```

The special literal # is added to the (only) positive clause $\{P, R\}$.

Iterative deepening

Prolog uses a simple depth-first search strategy to explore the search space, which is incomplete.[4] This kind of incompleteness would result in a calculus that hardly proves any formula. In order to obtain a complete proof search in the connection calculus, iterative deepening on the proof depth, i.e. the size of the active path, is performed. It is achieved by inserting the following lines into the code of Fig. 11:

```
(10) ( Grnd1=g -> true ; length(Path,K),
                          K<PathLim -> true ;
(11)   \+ pathlim -> assert(pathlim), fail ),
```

and adding the following lines to the `prove` predicate implementing the start rule:

```
(d)  prove(PathLim,Set) :-
(e)               % switch to complete strategy
(f)      retract(pathlim) ->
(g)      PathLim1 is PathLim+1,
                              prove(PathLim1,Set).
```

When the extension rule is applied and the new clause is not ground, i.e. it does not contain any variable, it is checked whether the size `K` of the active path exceeds the current path limit `PathLim` (line 10).[5] In this case the predicate `pathlim` is written into Prolog's database (line 11) indicating the need to increase the path limit if the proof search with the current path limit fails. If the proof search fails and the predicate `pathlim` can be found in the database (line f), then `PathLim` is increased and the proof search starts again (line g). Together with regularity (see Section 5.3) the resulting program is a decision procedure for ground (e.g. propositional) formulae and it is also able to refute some invalid first-order formulae.

[4]See, e.g., the query "`?-a.`" for the program "`a:-a. a.`".

[5]The *if-then-else* construct `Cond->Then;Else` succeeds if `Cond` and `Then` succeed, or if `Cond` fails and then `Else` succeeds.

5.2. Definitional clausal form

The definitional clausal-form transformation of Section 3.2 was implemented in Prolog as well. The complete translation consists of five steps:

(1) Renaming all term variables in the given formula.
(2) Transforming the formula into a Skolemized negation and/or definitional normal form.
(3) Transforming the negation and/or definitional normal form into a disjunctive normal form.
(4) Transforming the disjunctive normal form into a matrix.
(5) Reordering the clauses in the matrix (optional).

In the first step all term variables are renamed, so that each variable name occurs only once in the given formula. In the second step the formula is translated into a negation or definitional normal form according to Section 3.2. Additional options (see below) specify if the standard transformation or the definitional transformation is used. During this step *Skolemization* is performed as well: all universally quantified variables are substituted by a Skolem term (positive representation!) and universal and existential quantifiers are removed from the formula. The same Skolem term is used for instances of the same subformula. This is an optimization similar to the liberalized δ^{+}-rule for analytic tableaux [11]. As this kind of Skolemization can be motivated in an entirely proof-theoretical way,[6] it can also be adapted to, e.g., intuitionistic logic [26] (see Section 7.2). In a third step the formula is translated into disjunctive normal form and the fourth step transforms it into a matrix. Two simple optimizations are applied in this step as well. If a literal L occurs more than once in a clause, all syntactically identical duplicates of L are deleted. And if a clause contains two identical atoms with different polarities, e.g. P and $\neg P$, the clause is removed from the matrix. In an optional fifth step the clauses of the matrix are reordered using a simple perfect shuffle algorithm.

The clausal-form transformation is implemented by the main predicate

```
make_matrix(Fml,Matrix,Set)
```

where `Fml` is a first-order formula, `Matrix` is the returned matrix of the given formula, and `Set` is a

[6]Together with the occurs-check of the term unification, Skolemization is a technique to check if the reduction ordering is acyclic; see [4,45].

list of options. The syntax of the formula `Fml` is inductively defined as follows: a Prolog term, e.g. `p(f(c,X),g(Y))`, is a (atomic) formula; if `A` and `B` are formulae, then `(~A)` (negation), `(A;B)` (disjunction), `(A,B)` (conjunction), `(A => B)` (implication), `(A <=> B)` (equivalence), `(all X:A)` (universal quantifier) and `(ex X:A)` (existential quantifier) are formulae as well. The returned matrix is a list of clauses where each clause is a list of literals.

The following options can be included in the list `Set`: either `def` or `nodef`, `conj` and `reo(I)` where `I` is a natural number. The options `def` and `nodef` specify which transformation into clausal form is used:

(a) If none of the two options `def` or `nodef` are specified (*default transformation*): if the given formula has the form $A \Rightarrow C$, the standard transformation is applied to A (usually the axioms), while the definitional transformation is applied to (the conjecture) C; otherwise the definitional transformation is applied to the whole formula.
(b) If `def` is specified, the definitional transformation is applied to the whole formula.
(c) If `nodef` is specified, the standard transformation is applied to the whole formula.

If the option `conj` is included in `Set` and the given formula has the form $A \Rightarrow C$, then the special literal # is added to all clauses of the conjecture C to mark them as start clauses. Otherwise, the literal # is added to all positive clauses (see Section 5.1). If the option `reo(`I`)` is specified, all clauses of the final matrix are reordered I times using a perfect shuffle algorithm.

Example 13 (Definitional clausal form). Consider the following first-order formula from Example 1:

$$\big(((\exists x Q(x) \vee \neg Q(c)) \Rightarrow P) \\ \wedge (P \Rightarrow (\exists y Q(y) \wedge R))\big) \Rightarrow (P \wedge R).$$

It is translated into (the default) clausal form by calling the predicate `make_matrix((((((ex X:q(X)); (~q(c))) => p), (p => ((ex Y: q(Y)), r))) => (p,r)), M,[])`. It will return the matrix `M =[[p,r], [q(Z),-(p)],[-(q(c)),-(p)],[p,-(q(1^ []))],[p,-(r)]]`, in which `1^[]` is a (constant) Skolem term and `Z` is a new variable.

The complete source code of the clausal-form transformation is available on the leanCoP website.

5.3. Regularity and lemmata

The regularity condition of Section 3.3 is checked whenever the reduction, extension or lemma rule is applied. The substitution σ is not modified, i.e. the regularity condition is fulfilled if the open subgoal does not contain a literal that is syntactically identical with a literal in the active path. It is implemented by inserting the following line into the code of Fig. 11:

```
(3)  \+ (member(LitC,[Lit|Cla]),
        member(LitP,Path), LitC==LitP),
```

The Prolog predicate `\+` *Goal* succeeds only if *Goal* cannot be proven. In line 3 the corresponding *Goal* succeeds if the open subgoal `[Lit|Cla]` contains a literal `LitC` that is syntactically ("`==`") identical with a literal `LitP` in the active path `Path`. The (built-in) predicate `member` is used to enumerate all elements of a list. In leanCoP 1.0 a weaker form of regularity, called strictness [18], was implemented: no ground clause is used more than once on a branch.

The set of lemmata is represented by the list `Lem`. The lemma rule as described in Section 3.3 is then implemented by inserting the following lines:

```
(5)  ( member(LitL,Lem), Lit==LitL
(6)    ;
```

In order to apply the lemma rule the substitution σ is not modified, i.e. the lemma rule is only applied if the list of lemmata `Lem` contains a literal `LitL` that is syntactically identical with the literal `Lit`. Furthermore, the Literal `Lit` is added to the list `Lem` of lemmata in the (left) premise of the reduction and extension rule by adapting the following line:

```
(15)  prove(Cla,Path,PathLim,[Lit|Lem],Set).
```

In the resulting implementation the lemma rule is applied before the reduction and extension rules.

5.4. Restricted backtracking

According to Definition 8 in Section 4.3 backtracking is restricted by cutting off alternative rule applications once a solution for a literal is found. In Prolog the *cut* ("`!`") is used to cut off alternative solutions when Prolog tries to prove a goal. The Prolog cut is a built-in predicate, which succeeds immediately when first encountered as a goal. Any attempt to resatisfy the cut fails for the parent goal, i.e. other alternative choices are discarded that have been made from the point when the parent goal was invoked. Consequently, restricted backtracking is achieved by inserting a Prolog cut after the lemma, reduction, or extension rule is applied. It is implemented by inserting the following line into the code of Fig. 11:

```
(14) ( member(cut,Set) -> ! ; true ),
```

Restricted backtracking is switched on if the list `Set` contains the option `cut`. The restricted start step of Definition 8 in Section 4.3 cuts off alternative start clauses and is implemented by adapting the following lines of the start rule:

```
(b)  \+member(scut,Set) ->
           prove([-(#)],[],PathLim,[],Set) ;
(c)  lit(#,C,_) ->
           prove(C,[-(#)],PathLim,[],Set).
```

The restricted start step is used if the list `Set` includes the option `scut`. In this case (line c) the first clause `C` containing the special literal # is selected and the proof search starts with the open subgoal `C`; the active path is set to `{-#}` in order to solve literals # that might still be included in other start clauses. There is no backtracking for the goal `lit(#,C,_)` as it occurs in an *if-then-else* condition. Otherwise the proof search starts in the usual way (line b).

As pointed out in Section 4.3, restricted backtracking and the restricted start step lead to an incomplete proof search. In order to regain completeness, these strategies can be switched off when the search reaches a certain path limit. If the list `Set` contains the option `comp(`*Limit*`)`, where *Limit* is a natural number, the proof search is stopped and started again without using these incomplete search strategies. It is implemented by inserting the following lines:

```
(e)  member(comp(Limit),Set), PathLim=Limit
                                -> prove(1,[]) ;
(f)  (member(comp(_),Set);retract(pathlim)) ->
```

If the path limit reaches `Limit`, the proof search starts again with an empty set of options, i.e. a complete search strategy (line e). Until the `Limit` is reached iterative deepening continues even if the current path limit is not exceeded during the proof search (line f). This is necessary to allow the incomplete strategies, for which the path limit during the incomplete search might not be exceeded, to reach the path limit `Limit` and to start a complete proof search.

5.5. Strategy scheduling

Different options in the list `Set` are used to control the proof search. They determine if, e.g., a definitional clausal-form transformation or restricted backtracking should be used. The setting or *strategy* `Set` is a list of options that is either empty or contains one or more of the following options:

(1) `nodef/def`: The standard (`nodef`) or definitional (`def`) transformation into clausal form is carried out. If none of these two options is specified, the default transformation is used (see Section 5.2).
(2) `conj`: The conjecture clauses of the formula are used as start clauses (see Section 5.2).
(3) `reo(`I`)`: The clauses in the matrix are reordered I times (see Section 5.2).
(4) `scut`: The restricted start step is used (see Sections 4.3 and 5.4).
(5) `cut`: Restricted backtracking is used (see Sections 4.3 and 5.4).
(6) `comp(`I`)`: The options `scut` and `cut` are switched off when iterative deepening exceeds the path limit I (see Section 5.4).

The option `conj` is complete only for formulae with a provable conjecture[7] and `scut` as well as `cut` are complete only if used in combination with `comp(`I`)`.

leanCoP 2.0 uses a fixed strategy scheduling. The leanCoP 2.0 core prover shown in Fig. 12 is consecutively invoked by a shell script with different strategies, each for a specific time. This increases the chance to find a proof for a given problem, since most strategies are only appropriate for certain kinds of problems. For example, the definitional clausal-form transformation works well for problems that are not in clausal form. But for problems that are "almost" in clausal form this transformation might have a negative effect on the proof search.

Practical evaluations suggest using the following scheduling. The four strategies `[cut,comp(7)]`, `[conj,cut]`, `[def,scut,cut]` and `[nodef, scut,cut]` are each invoked for 2%, 60%, 16% and 4% of the total time limit, respectively. The next five strategies, each invoked for 2% of the total time limit, are similar to the second and third strategies but have the `reo` option added. For the remaining time the complete strategy `[]` is invoked. As this last strategy is complete the whole proof search is complete as well (with respect to an arbitrary large total time limit).

[7] See, e.g., the valid formula $(P \wedge \neg P) \Rightarrow Q$, for which there is no connection proof that starts with the clause $\{Q\}$.

```
prove(PathLim,Set) :-
    \+member(scut,Set) -> prove([-(#)],[],PathLim,[],Set) ;
    lit(#,C,_) -> prove(C,[-(#)],PathLim,[],Set).
prove(PathLim,Set) :-
    member(comp(Limit),Set), PathLim=Limit -> prove(1,[]) ;
    (member(comp(_),Set);retract(pathlim)) ->
    PathLim1 is PathLim+1, prove(PathLim1,Set).

prove([],_,_,_,_).
prove([Lit|Cla],Path,PathLim,Lem,Set) :-
    \+ (member(LitC,[Lit|Cla]), member(LitP,Path), LitC==LitP),
    (-NegLit=Lit;-Lit=NegLit) ->
       ( member(LitL,Lem), Lit==LitL
         ;
         member(NegL,Path), unify_with_occurs_check(NegL,NegLit)
         ;
         lit(NegLit,Cla1,Grnd1),
         ( Grnd1=g -> true ; length(Path,K), K<PathLim -> true ;
           \+ pathlim -> assert(pathlim), fail ),
         prove(Cla1,[Lit|Path],PathLim,Lem,Set)
       ),
       ( member(cut,Set) -> ! ; true ),
       prove(Cla,Path,PathLim,[Lit|Lem],Set).
```

Fig. 12. The complete source code of the leanCoP 2.0 core prover.

6. Performance

At first the performance of different clausal-form transformations is evaluated. Afterwards the impact of the different pruning techniques described in Sections 3 and 4 on the performance of leanCoP 2.0 is analysed. Finally leanCoP 2.0 is compared with other well-known automated theorem proving (ATP) systems. For the tests all 5051 non-clausal, so-called *FOF* problems of version 3.7.0 of the TPTP library [40] are considered. For the comparison of leanCoP 2.0 with other ATP systems all 6348 clausal, so-called *CNF* problems of version 3.7.0 of the TPTP library are considered as well.

Some of these problems do not have a conjecture and are either satisfiable or unsatisfiable. leanCoP and some other ATP systems do only determine if a given formula is valid or invalid. Since a formula F is unsatisfiable if, and only if, $\neg F$ is valid, these formulae are negated in order to determine if they are unsatisfiable or satisfiable. For the ATP systems that do not have built-in equality, e.g. leanCoP or lean*TAP*, the equality axioms are added to the problem formula using the TPTP2X tool, which is included in the TPTP library. All tests were performed on a 3 GHz Xeon system with 4 GB of RAM running Linux and ECLiPSe Prolog version 5.10. The time limit for all tests is 600 s.

6.1. *Comparing different clausal-form transformations*

Table 4 shows the results of the leanCoP 2.0 core prover (using the options `[cut,comp(7)]`) on different clausal form transformations. The following clausal-form transformations are evaluated: the clausal-form transformation of the TPTP2X tool (using the option `-t clausify:tptp`) of version 3.7.0 of the TPTP library (which uses an algorithm combining features of the Otter and the Quaife clausal transformations), the FLOTTER clausal-form transformation of SPASS 3.0 [43], the clausal-form transformation of E 1.0 [35], and the leanCoP 2.0 clausal-form transformation using the default transformation as well as the `def` and the `nodef` options (see Section 5.2).

The rows of the table show: the total number (and percentage) of proved problems, the number of problems proved within a certain time, the number and percentage of proved problems within a certain difficulty rating, the number of proved problems containing no equality and containing equality, the number of proved pure equality problems containing only equality (these problems are included in the row "With equality" as well), the number of refuted problems (i.e. non-theorems), the number of problems for which the time limit is exceeded, and the number of problems

Table 4

TPTP benchmark results for different clausal-form transformations

	TPTP	FLOTTER	E	leanCoP 2.0		
	3.7.0	3.0	1.0	"def"	"nodef"	(default)
Proved	1205	1365	1369	1486	1514	**1560**
[%]	24%	27%	27%	29%	30%	**31%**
0–1 s	958	1072	1068	1144	1201	**1230**
1–10 s	119	136	133	163	148	**144**
10–100 s	84	91	104	112	101	**109**
100–600 s	44	66	64	67	64	**77**
Rating 0.0	481	499	503	529	522	**531**
Rating >0.0	724	866	866	957	992	**1029**
Rating 0.00 … 0.24	53%	56%	58%	62%	60%	**62%**
Rating 0.25 … 0.49	39%	47%	47%	52%	51%	**53%**
Rating 0.50 … 0.74	10%	16%	16%	17%	22%	**24%**
Rating 0.75 … 1.00	1%	1%	1%	1%	2%	**2%**
No equality	539	552	559	590	582	**587**
With equality	666	813	810	896	932	**973**
Pure equality	13	23	13	27	13	**27**
Refuted	35	53	36	33	36	35
Time out	3137	3101	3115	3099	2958	3017
Error	674	532	531	433	543	439

that produce an error. The TPTP rating [41] expresses the relative difficulty of the problems from 0.0 (easy) to 1.0 (very difficult). The error row includes problems that produce stack overflows or memory allocation errors. It also includes 102 problems for which the TPTP2X tool could not generate the leanCoP format (due to their huge size). The time needed for the clausal transformation is included in the timings for the leanCoP transformation but is not included in the timings for the TPTP, FLOTTER and E transformations.

The default transformation of leanCoP 2.0, where the standard transformation is applied to the axioms and the definitional transformation is applied to the conjecture, shows the best performance. The leanCoP standard transformation solves slightly more problems than the definitional transformation of leanCoP (both applied to the whole formula). The definitional transformation still proves 86 problems not solved by the default transformation.

The performance of the FLOTTER and of the E transformation are almost identical and better than the TPTP transformation. These transformations might be better suited for saturation-based proof calculi such as resolution. For a better performance these transformations might also require the application of subsumption, which is a basic technique of ATP systems based on resolution, but not used in leanCoP at all.

6.2. *Comparing different pruning techniques*

Table 5 contains the results for the following variants of the leanCoP prover: leanCoP 1.0 [28], the "basic" version of the leanCoP 2.0 core prover described in Section 5.1, the "define" version enhanced by the (default) definitional clausal-form transformations as described in Section 5.2, the "regular" version that adds regularity and lemmata as described in Section 5.3, the "restrict" version that adds restricted backtracking as described in Section 5.4 and performs a complete search from path limit seven, and the actual leanCoP 2.0 prover, which uses strategy scheduling as described in Section 5.5. The "restrict" version consists of the leanCoP 2.0 core prover using the options `[cut,comp(7)]`. The rows of Table 5 were already explained in Section 6.1. An additional row shows the average proof time for the set of problems that are proved by all listed prover variants.

As a result of the lean Prolog technology (see Section 5.1), the "basic" version is in general about five times faster than leanCoP 1.0 (see row "Average time"). But it solves fewer problems since it does not use the strictness condition (see Section 5.3). The "define" version proves 46 problems not solved by the "basic" version. But 38 problems are not proved any-

Table 5

TPTP benchmark results for different techniques of leanCoP 2.0

	leanCoP 1.0	basic	define	regular	restrict	leanCoP 2.0
Proved	1105	1086	1094	1256	**1560**	1797
[%]	22%	22%	22%	25%	**31%**	36%
0–1 s	861	866	867	972	**1230**	1220
1–10 s	95	92	100	122	**144**	133
10–100 s	87	76	79	106	**109**	250
100–600 s	62	52	48	56	**77**	194
Average time	12.2 s	2.6 s	2.8 s	3.6 s	**2.6** s	6.1 s
Rating 0.0	458	450	446	501	**531**	554
Rating >0.0	647	636	648	755	**1029**	1243
Rating 0.00 ... 0.24	51%	50%	50%	57%	**62%**	67%
Rating 0.25 ... 0.49	37%	37%	36%	41%	**53%**	63%
Rating 0.50 ... 0.74	4%	4%	6%	7%	**24%**	33%
Rating 0.75 ... 1.00	0%	0%	0%	0%	**2%**	4%
No equality	532	526	515	552	**587**	616
With equality	573	560	579	704	**973**	1181
Pure equality	13	7	19	27	**27**	29
Refuted	1	14	10	35	35	35
Time out	3425	3432	3505	3321	3017	2501
Error	520	519	442	439	439	718

more. Even though this is only a modest improvement, the definitional transformation works very well in conjunction with restricted backtracking (see Section 6.2). The "regular" version solves 173 problems not solved by the "define" version. The "restrict" version (using restricted backtracking) shows the biggest improvement, in particular for problems that have a higher rating (rows ">0.0" and "Rating 0.50 ... 0.74") or that contain equality (row "With equality"). The "restrict" version solves 330 problems not solved by the "regular" version. A similar improvement can be seen for the final leanCoP 2.0 prover using strategy scheduling.

6.3. *Comparing* leanCoP *with other ATP systems*

In Table 6 the performance of leanCoP 2.0 on the FOF problems of the TPTP library is compared with the performance of the ATP systems lean*TAP* [2] (the first popular lean theorem prover), leanCoP 1.0 [28] (the first version of leanCoP), SETHEO 3.3[8] [16] (one of the fastest connection provers), Otter 3.3 [21,22] (still used as the standard benchmark), version "2009-02A" of Prover9 [23] (the successor of Otter) and E 1.0 ("Temi") [35] (one of the leading ATP systems).

In addition to the rows already shown in Tables 4 and 5, the number of proved problems for each problem domain [40] is given. The "Error" row now also contains problems on which an ATP system gave up. For example, Otter and Prover9 often gave up because of an empty set-of-support.

leanCoP 2.0 proves significantly more problems than leanCoP 1.0, Otter and SETHEO. One notices again a high number of solved problems that are rated difficult (row "Rating 0.50 ... 0.74") or that contain equality (row "With equality"). Note that leanCoP has no built-in inference rules for equality. leanCoP 2.0 proves more problems of the AGT and the NUM domain than E. Its performance is similar to that of E, e.g., in the domains CAT, GEO, KRS, MED, MSC, SET and SEU, but significantly lower for problems in, e.g., the domains ALG, LCL and SWC. The tableau prover lean*TAP* shows a good performance for easy problems, e.g. in the SYN domain, but does not perform very well on larger, more difficult problems, e.g. problems in the domains NUM, SET or SEU.

It general leanCoP 2.0 performs better than E on problems where the goal-directed approach of the underlying connection calculus is more likely able to

[8]For SETHEO the options -dr (iterative deepening), -reg (regularity) and -st (subsumption and tautology) were used, which showed the best performance.

Table 6
TPTP benchmark results for leanCoP and other ATP systems – FOF problems

	lean*TAP*	leanCoP	SETHEO	OTTER	Prover9	leanCoP	E
	2.3	1.0	3.3	3.3	2009-02A	2.0	1.0
Proved	405	1105	1296	1389	1664	**1797**	2541
[%]	8%	22%	26%	27%	33%	**36%**	50%
0–1 s	379	861	941	1064	1285	**1220**	1912
1–10 s	13	95	217	184	200	**133**	258
10–100 s	12	87	73	107	126	**250**	270
100–600 s	1	62	65	34	53	**194**	101
Rating 0.0	228	458	497	507	450	**554**	610
Rating >0.0	177	647	799	882	1214	**1243**	1931
Rating 0.00 ... 0.24	17%	51%	57%	64%	61%	**67%**	75%
Rating 0.25 ... 0.49	18%	37%	46%	47%	71%	**63%**	92%
Rating 0.50 ... 0.74	2%	4%	8%	3%	27%	**33%**	74%
Rating 0.75 ... 1.00	0%	0%	0%	0%	1%	**4%**	12%
No equality	319	532	549	535	497	**616**	697
With equality	86	573	747	854	1167	**1181**	1844
Pure equality	12	13	13	47	69	**29**	168
AGT	0	17	17	16	17	**24**	20
ALG	11	14	18	61	86	**34**	173
BOO	0	0	0	0	0	**0**	0
CAT	0	1	0	1	0	**3**	4
COM	0	1	3	3	6	**4**	6
CSR	15	84	85	63	27	**136**	210
GEO	23	143	159	160	171	**171**	174
GRA	0	4	6	5	9	**6**	15
GRP	1	6	5	7	14	**9**	21
HAL	0	0	2	1	0	**1**	4
KRS	32	70	89	106	103	**105**	112
LAT	0	2	3	3	30	**15**	29
LCL	3	26	32	18	45	**24**	80
MED	0	0	1	5	1	**7**	9
MGT	11	31	41	54	60	**45**	67
MSC	1	2	2	2	2	**3**	3
NLP	3	3	7	6	11	**13**	22
NUM	1	34	58	30	43	**60**	58
PLA	0	0	0	0	0	**0**	0
PUZ	2	5	6	6	7	**7**	10
SET	22	160	187	214	247	**318**	324
SEU	8	141	143	170	259	**329**	359
SWC	14	14	66	84	98	**81**	325
SWV	55	142	154	157	178	**177**	225
SYN	201	200	205	211	239	**217**	278
TOP	2	5	7	6	11	**8**	13
Refuted	0	1	27	0	0	35	372
Time out	3502	3077	2510	681	1502	2501	2138
Error	1144	868	1218	2981	1885	718	0

find a proof. leanCoP 2.0 performs in general better than leanCoP 1.0 and SETHEO on problems that contain many axioms and/or equality axioms. leanCoP 2.0 proves 506 problems not proved by Prover9 and 181 problems not proved by E. Conversely, Prover9 proves 378 problems and E proves 930 problems not proved by leanCoP 2.0. As Prover9 is tuned towards algebraic problems, it solves much more problems of the ALG domain than leanCoP 2.0. iProver 0.5 [12] and Vampire 10.0 [42], which were together with E among the top ATP systems at the CASC-J4 system competition [39], prove 2299 and 2699 problems, respectively.

Table 7 shows the performance results on the CNF problems of the TPTP library for leanCoP 2.0 and the other ATP systems of Table 6. For leanCoP 2.0 the TPTP2X tool was used to transform all clausal-form problems of the TPTP library into the first-order format (using the option `-t fofify:obvious`).

Again the performance of leanCoP 2.0 has improved compared to leanCoP 1.0. But its relative performance compared to SETHEO, Otter, Prover9 and E is not as good as for the FOF problems. This might be due to the fact that these ATP systems are better tuned to the CNF problems of the TPTP library. Restricted backtracking, which works well in conjunction with the definitional clausal-form transformation of leanCoP 2.0, might have a less significant effect as well. iProver 0.5 and Vampire 10.0 prove 2743 and 3652 CNF problems, respectively.

7. Improvements and extensions

The compact style of the leanCoP 2.0 core prover makes it an ideal starting point for further improvements and extensions. This section describes how the proof search order of leanCoP can be randomized, how leanCoP can be extended to deal with intuitionistic logic, and how the leanCoP core prover performs on different Prolog systems.

7.1. Randomizing the proof search order

Since restricted backtracking cuts off alternative connections (see Section 4), it might cut off some connections required for a proof. Therefore the benefit of this approach strongly depends on the proof search order. The proof search order, in turn, usually depends on the order of clauses and literals in the given formula. Whereas one order of clauses might be ideal to quickly find a proof, it might be impossible to find a proof for another order of the same clauses.

randoCoP [32] extends the leanCoP 2.0 implementation. It repeatedly: (a) reorders the axioms and literals

Table 7
TPTP benchmark results for leanCoP and other ATP systems – CNF problems

	lean*TAP*	leanCoP	SETHEO	leanCoP	Otter	Prover9	E
	2.3	1.0	3.3	2.0	3.3	2009-02A	1.0
Proved	278	1391	1843	**1906**	2635	2966	3969
[%]	4%	22%	29%	**30%**	42%	47%	63%
0–1 s	248	957	1476	**1362**	2068	2320	2971
1–10 s	16	208	128	**172**	293	338	634
10–100 s	5	157	169	**226**	192	179	271
100–600 s	9	69	70	**146**	82	129	93
Rating 0.0	249	974	1175	**1209**	1595	1583	1666
Rating >0.0	29	417	668	**697**	1040	1383	2303
Rating 0.00 ... 0.24	9%	41%	52%	**53%**	77%	80%	85%
Rating 0.25 ... 0.49	1%	10%	17%	**19%**	22%	37%	71%
Rating 0.50 ... 0.74	0%	6%	9%	**10%**	5%	19%	71%
Rating 0.75 ... 1.00	0%	0%	0%	**1%**	0%	1%	13%
No equality	277	932	1058	**1082**	1119	1145	1412
With equality	1	459	785	**824**	1516	1821	2557
Pure equality	1	136	201	**166**	627	833	954
Refuted	0	6	23	109	0	0	423
Time out	5658	4896	4345	4328	0	976	1956
Error	412	55	137	5	3713	2406	0

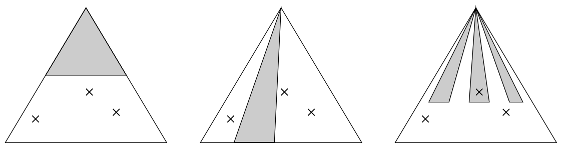


Fig. 13. Search strategies: Complete, restricted backtracking, restricted backtracking with reordering.

of a given problem at random and (b) invokes the leanCoP 2.0 core prover. This increases the chance to find a proof, in particular for the incomplete search strategies. Figure 13 illustrates this search strategy. The triangle represents the search space, the crosses mark the solutions, and the grey shaded area is the search space that can be traversed within a certain time limit and is roughly the same for all three triangles. The complete search strategy (left-hand side) does not reach the search depth required for a solution. The search with restricted backtracking (in the middle) reaches the depth of the solutions but narrows the search space too much and does not reach the required breadth. Only the search strategy with restricted backtracking and a repeated reordering of the axioms/clauses (right-hand side) is able to find a proof. leanCoP 2.0 already contains an option for reordering clauses (see Section 5.5). But the effect on the proof search is rather small, since the generated clause orders are not sufficiently diverse. A random reordering mixes the order of clauses more thoroughly.

randoCoP outputs a readable connection proof. An additional argument is added to the `prove` predicate of the leanCoP 2.0 core prover that records a compact connection (tableau) proof. This compact connection proof is then converted into a readable proof. randoCoP uses an additional module that translates problems represented in the TPTP syntax into the leanCoP syntax. This module also adds the required equality axioms.

The result of running randoCoP on all first-order problems of version 3.7.0 of the TPTP library is shown in Table 8. It proves more problems than leanCoP 2.0 in the domains AGT (36 proved problems), CSR (162), NUM (70) and SEU (352). At the CASC-J4 system competition randoCoP was ranked third out of 11 ATP systems in the most important FOF division that output a proof [39].

7.2. *Intuitionistic logic*

The matrix characterization of classical validity (see Lemma 1) can be extended to some non-classical logics, such as modal or intuitionistic logic [45]. To this end a so-called prefix, i.e. a string consisting of variables and constants, which essentially encodes the *Kripke world* semantics, is assigned to each literal. For a complementary connection $\{L_1 : p_1, L_2 : p_2\}$ not only the terms of both literals need to unify under a term substitution σ, i.e. $\sigma(L_1) = \sigma(\overline{L_2})$, but also the corresponding prefixes p_1 and p_2 are required to unify under a prefix substitution σ', i.e. $\sigma'(p_1) = \sigma'(p_2)$.

ileanCoP is an automated theorem prover for intuitionistic first-order logic and implements a connection calculus for intuitionistic logic, which adds a prefix to each literal and is based on a *clausal* version of the matrix characterization for intuitionistic logic [26]. It uses the classical search engine of leanCoP and an additional prefix unification algorithm [29] to unify the prefixes of the literals in every connection. This ensures that the characteristics of intuitionistic logic are respected and the given formula is intuitionistically valid (see also [13,44,45]). As the intuitionistic characteristics are captured in a separate prefix substitution, all techniques and inference rules presented in Sections 3 and 4 can be adapted to work with the intuitionistic calculus, e.g. the definitional clausal from translation and regularity. Restricted backtracking can directly be used without any modifications.

ileanCoP 1.2 [27] enhances ileanCoP 1.0 by integrating these new inference rules and search techniques. Only a few additions are necessary to turn the leanCoP 2.0 core prover into the ileanCoP 1.2 core prover; see [27] for details. The prefix unification algorithm requires another 26 lines of Prolog code; see [25, 29] for details. The full source code of ileanCoP 1.2 is available on the leanCoP website.

ileanCoP 1.2 proves around three times more problems of version 3.3.0 of the TPTP library than any other ATP system for intuitionistic logic [27]. A comprehensive evaluation of intuitionistic ATP systems is also available in the ILTP library [33].[9] The result of running ileanCoP 1.2 on all first-order problems of version 3.7.0 of the TPTP library is shown in Table 8. Though theorem proving in intuitionistic logic is considered much more difficult than in classical logic,[10] ileanCoP 1.2 proves almost as many problems as, e.g., SETHEO. It proves more problem than Prover9 in the domains AGT (18 proved problems), CAT (1), CSR (133), HAL (1), MED (3) and NUM (55).

[9]See the ILTP library website at http://www.iltp.de.

[10]Deciding if a propositional formula is valid is $co - \mathcal{NP}$-complete for classical logic [6], but $\mathcal{PSPACE}$-complete for intuitionistic logic [36].

Table 8
TPTP benchmark results for randoCoP, ileanCoP and different Prolog systems

	randoCoP	leanCoP	leanCoP 2.0 core			ileanCoP
	1.1	2.0	SWI-Prolog	SICStus	ECLiPSe	1.2
Proved	1827	1797	1603	1602	1548	1272
[%]	36%	36%	32%	32%	31%	25%
0–1 s	1223	1220	1185	1201	1196	851
1–10 s	268	133	156	173	157	101
10–100 s	229	250	163	148	124	100
100–600 s	107	194	99	80	71	220
Average time	2.7 s	13.1 s	5.0 s	4.0 s	4.3 s	72.9 s
Rating 0.0	546	554	533	534	530	480
Rating >0.0	1281	1243	1070	1068	1018	792
Rating 0.00 ... 0.24	67%	67%	62%	63%	61%	55%
Rating 0.25 ... 0.49	64%	63%	55%	54%	52%	37%
Rating 0.50 ... 0.74	38%	33%	26%	25%	23%	18%
Rating 0.75 ... 1.00	4%	4%	3%	3%	3%	1%
No equality	632	616	586	584	583	494
With equality	1195	1181	1017	1018	965	778
Pure equality	26	29	28	29	28	19
Refuted	30	35	35	35	35	71
Time out	3087	2501	3348	3360	2968	2684
Error	107	718	65	54	500	1024

7.3. *Evaluating different Prolog systems*

For leanCoP the performance and stability of the used Prolog system have an impact on its performance as well. In order to evaluate the performance of different Prolog systems, the leanCoP 2.0 core prover (using options `[cut,comp(7)]`) was tested with the following Prolog systems: ECLiPSe 5.10, SICStus 4.0.4 and SWI-Prolog 5.6.59.[11] For these tests an additional Prolog module is used that translates from the TPTP problem syntax into the leanCoP syntax and also adds the required equality axioms. The time for this translation is included in the proof time.

The results of running the leanCoP 2.0 core prover on all FOF problems of version 3.7.0 of the TPTP library are shown in Table 8. The performance of the three Prolog systems is essentially similar (see the "average time" row), but its *static* symbol table causes the ECLiPSe system to crash on many of the huge problems (500 errors compared to 54/65 errors for SICStus/SWI-Prolog).

[11]More information about these Prolog systems can be found on the websites http://www.eclipse-clp.org (ECLiPSe), http://www.sics.se/isl/sicstuswww/site (SICStus), and http://www.swi-prolog.org (SWI-Prolog).

8. Conclusion

Proof search in connection calculi requires in general a large amount of backtracking. Limiting this backtracking is crucial for a more effective proof search. *Restricted backtracking* is a simple technique for reducing the amount of backtracking significantly. Though it is not a complete search strategy, it performs very well in practice, in particular for problems that contain many axioms. A definitional transformation into clausal form was presented and shown to work well with connection calculi, as it focuses on the number of connections instead of the number of clauses. Regularity, lemmata and restricted backtracking are the basis for a refined connection calculus. It was shown how the basic connection calculus together with the described techniques and inference rules can be converted step by step into a compact Prolog implementation. Together with a fixed strategy scheduling, it is the basis of the leanCoP 2.0 prover.[12]

The usefulness of the presented techniques has been empirically evaluated. Comprehensive tests were run

[12]The source code of leanCoP 2.0 and more information is available on the leanCoP website at http://www.leancop.de.

on all problems of the most recent version of the TPTP library. The integration of regularity into the basic calculus improves performance significantly. It confirms the observation that regularity is one of the most successful techniques for pruning the search space in connection calculi [18]. The performance improvement of restricted backtracking seems to be even greater, in particular for problems that contain equality or a large number of axioms. Thus, restricted backtracking is currently the single most effective technique for pruning the search space in connection calculi. The presented definitional clausal-form transformation yields better results than other well-known transformations. Alltogether this leads to a significant performance improvement of leanCoP 2.0 compared to leanCoP 1.0.

To sum up the main results of this paper:

- Restricting backtracking is crucial for an effective proof search in connection calculi.
- Clausal-form transformations for connection calculi need to consider the number of possible connections; transformations that are optimized for saturation-based calculi might not work as well for connection calculi.
- The implementation language as well as the size and complexity of an ATP system is not essential for its performance.

The strong dependency of restricted backtracking on the order of clauses can be reduced by randomly reordering the axioms and clauses, as implemented in randoCoP [32]. Restricted backtracking works well for some other logics as well. By just adding prefixes and an additional prefix unification algorithm, the implementation is turned into the theorem prover ileanCoP for intuitionistic first-order logic [26,27].

For problems that are already in clausal form and for some first-order problem domains of the TPTP library, some state-of-the-art ATP systems still solve many more problems than leanCoP. To further improve performance, additional techniques and strategies need to be developed. The presented refined connection calculus and its compact implementation are an ideal starting point for the integration and evaluation of such search techniques. Possible techniques include, e.g., the folding-up rule [18] and the selection of a strategy according to the specific characteristics of a given problem. It also needs to be investigated if and how backtracking can be restricted in a way, such that completeness is retained.

Future research also includes the integration of arithmetic into the leanCoP implementation and the extension of the calculus and of the implementation to other non-classical logics, e.g. modal logics, that are considered within the matrix characterization framework [13, 30,44,45].

Acknowledgements

The author would like to thank Thomas Raths for providing the benchmark statistics for all tested ATP systems and for testing the different clausal-form transformations. The author would also like to thank Christoph Kreitz, Paul Milkaitis and Stephan Schmitt for their helpful feedback and the anonymous referees for their comprehensive and useful comments.

References

[1] P. Baumgartner, N. Eisinger and U. Furbach, A confluent connection calculus, in: *16th CADE*, H. Ganzinger, ed., LNAI, Vol. 1632, Springer, Heidelberg, 1999, pp. 329–343.

[2] B. Beckert and J. Posegga, lean*TAP*: lean, tableau-based theorem proving, in: *12th CADE*, A. Bundy, ed., LNAI, Vol. 814, Springer, Heidelberg, 1994, pp. 793–797.

[3] W. Bibel, Matings in matrices, *Communications of the ACM* **26** (1983), 844–852.

[4] W. Bibel, *Automated Theorem Proving*, Vieweg, Wiesbaden, 1987.

[5] W. Clocksin and C. Mellish, *Programming in Prolog*, Springer, Heidelberg, 1981.

[6] S.A. Cook, The complexity of theorem-proving procedures, in: *Proceedings of 3rd Annual ACM Symposium on the Theory of Computing*, Shaker Heights, OH, USA, 1971, pp. 151–158.

[7] E. Eder, *Relative Complexities of First Order Calculi*, Vieweg, Wiesbaden, 1992.

[8] M.C. Fitting, *First-order Logic and Automated Theorem Proving*, Springer, Heidelberg, 1990.

[9] G. Gentzen, Untersuchungen über das logische Schließen, *Mathematische Zeitschrift* **39** (1935), 176–210, 405–431.

[10] R. Hähnle, Tableaux and related methods, in: *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, eds, Elsevier, Amsterdam, 2001, pp. 100–178.

[11] R. Hähnle and P. Schmitt, The liberalized δ-rule in free variable semantic tableaux, *Journal of Automated Reasoning* **13** (1994), 211–221.

[12] K. Korovin, iProver – An instantiation-based theorem prover for first-order logic (system description), in: *IJCAR'2008*, A. Armando, P. Baumgartner and G. Dowek, eds, LNCS, Vol. 5195, Springer, Heidelberg, 2008, pp. 292–298.

[13] C. Kreitz and J. Otten, Connection-based theorem proving in classical and non-classical logics, *Journal of Universal Computer Science* **5** (1999), 88–112.

[14] S.-J. Lee and D. Plaisted, Eliminating duplicates with the hyper-linking strategy, *Journal of Automated Reasoning* **9** (1992), 25–42.

[15] R. Letz, Properties and relations of tableau and connection calculi, in: *Intellectics and Computational Logic*, S. Hölldobler, ed., Kluwer, Amsterdam, 2000, pp. 245–261.

[16] R. Letz, J. Schumann, S. Bayerl and W. Bibel, SETHEO: a high-performance theorem prover, *Journal of Automated Reasoning* **8** (1992), 183–212.

[17] R. Letz, K. Mayr and C. Goller, Controlled integration of the cut rule into connection tableaux calculi, *Journal of Automated Reasoning* **13** (1994), 297–337.

[18] R. Letz and G. Stenz, Model elimination and connection tableau procedures, in: *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, eds, Elsevier, Amsterdam, 2001, pp. 2015–2114.

[19] D. Loveland, Mechanical theorem proving by model elimination, *Journal of the ACM* **15** (1968), 236–251.

[20] A. Martelli and U. Montanari, An efficient unification algorithm, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **4** (1982), 258–282.

[21] W. McCune, OTTER 2.0. in: *CADE-10*, M.E. Stickel, ed., LNCS, Vol. 449, Springer, Heidelberg, 1990, pp. 663–664.

[22] W. McCune, OTTER 3.0 reference manual and guide, Technical Report ANL-94/6, Argonne National Laboratory, 1994.

[23] W. McCune, Release of Prover9, in: *Mile High Conference on Quasigroups, Loops and Nonassociative Systems*, Denver, 2005.

[24] A. Nonnengart and C. Weidenbach, Computing small clause normal forms, in: *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, eds, Elsevier, Amsterdam, 2001, pp. 335–367.

[25] J. Otten, ileanTAP: an intuitionistic theorem prover, in: *TABLEAUX'97*, D. Galmiche, ed., LNAI, Vol. 1227, Springer, Heidelberg, 1997, pp. 307–312.

[26] J. Otten, Clausal connection-based theorem proving in intuitionistic first-order logic, in: *TABLEAUX'2005*, B. Beckert, ed., LNAI, Vol. 3702, Springer, Heidelberg, 2005, pp. 245–261.

[27] J. Otten, leanCoP 2.0 and ileanCoP 1.2: high performance lean theorem proving in classical and intuitionistic logic, in: *IJCAR'2008*, A. Armando, P. Baumgartner and G. Dowek, eds, LNCS, Vol. 5195, Springer, Heidelberg, 2008, pp. 283–291.

[28] J. Otten and W. Bibel, leanCoP: lean connection-based theorem proving, *Journal of Symbolic Computation* **36** (2003), 139–161.

[29] J. Otten and C. Kreitz, T-string-unification: unifying prefixes in non-classical proof methods, in: *TABLEAUX'96*, P. Miglioli, U. Moscato, D. Mundici and M. Ornaghi, eds, LNAI, Vol. 1071, Springer, Heidelberg, 1996, pp. 244–260.

[30] J. Otten and C. Kreitz, A uniform proof procedure for classical and non-classical logics, in: *KI-96: Advances in Artificial Intelligence*, G. Görz and S. Hölldobler, eds, LNAI, Vol. 1137, Springer, Heidelberg, 1996, pp. 307–319.

[31] D. Plaisted and S. Greenbaum, A structure-preserving clause form translation, *Journal of Symbolic Computation* **2** (1986), 293–304.

[32] T. Raths and J. Otten, randoCoP: randomizing the proof search order in the connection calculus, in: *IJCAR'08 Workshop on Practical Aspects of Automated Reasoning (PAAR-2008)*, Sydney, Australia, B. Konev, R. Schmidt and S. Schulz, eds, CEUR Workshop Proceedings, 2008, pp. 94–102.

[33] T. Raths, J. Otten and C. Kreitz, The ILTP problem library for intuitionistic logic, *Journal of Automated Reasoning* **38** (2007), 261–271.

[34] J.A. Robinson, A machine-oriented logic based on the resolution principle, *Journal of the ACM* **12**(1) (1965), 23–41.

[35] S. Schulz, E – a Brainiac theorem prover, *AI Communications* **15**(2) (2002), 111–126.

[36] R. Statman, Intuitionistic propositional logic is polynomial-space complete, *Theoretical Computer Science* **9** (1979), 67–72.

[37] M. Stickel, A Prolog technology theorem prover: implementation by an extended Prolog compiler, *Journal of Automated Reasoning* **4** (1988), 353–380.

[38] M. Stickel, A Prolog technology theorem prover: a new exposition and implementation in Prolog, *Theoretical Computer Science* **104** (1992), 109–128.

[39] G. Sutcliffe, The 4th IJCAR automated theorem proving system competition, *AI Communications* **22** (2009), 59–72.

[40] G. Sutcliffe and C. Suttner, The TPTP problem library – CNF release v1.2.1, *Journal of Automated Reasoning* **21** (1998), 177–203.

[41] G. Sutcliffe and C. Suttner, Evaluating general purpose automated theorem proving systems, *Artificial Intelligence* **131**(1,2) (2001), 39–54.

[42] A. Riazanov and A. Voronkov, The design and implementation of Vampire, *AI Communications* **15**(2,3) (2002), 91–110.

[43] C. Weidenbach, R. Schmidt, T. Hillenbrand, R. Rusev and D. Topic, System description: SPASS version 3.0, in: *CADE-21*, F. Pfenning, ed., LNCS, Vol. 4603, Springer, Heidelberg, 2007, pp. 514–520.

[44] A. Waaler, Connections in nonclassical logics, in: *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, eds, Elsevier, Amsterdam, 2001, pp. 1487–1578.

[45] L. Wallen, *Automated Deduction in Nonclassical Logic*, MIT Press, Cambridge, 1990.