20 Years of leanCoP — An Overview of the Provers

Jens Otten*

Department of Informatics, University of Oslo, Norway

Abstract

This paper gives a comprehensive overview of the automated theorem prover leanCoP and all its variants for classical and non-classical first-order logics that have been developed so far. It includes historical details describing how seven lines of Prolog code turned into some of the most popular and efficient connection provers for classical and non-classical logics. The paper provides an overview of the non-clausal version nanoCoP and other implementations inspired by leanCoP.

Keywords

leanCoP, connection calculus, classical logic, automated reasoning, non-classical logics

1. Introduction

Automating *formal reasoning* is a core research field in *Artificial Intelligence*. One of its main goals is to develop efficient *(theorem) provers* that automate the search for a formal proof of a given conjecture with respect to a specific logic.

20 years ago, the article "leanCoP: Lean Connection-based Theorem Proving" [1] started the development of a series of compact theorem provers for classical and several non-classical logics. The very first version of leanCoP was written a few years earlier, in November 1999, when the author was asked by Wolfgang Bibel to take on a lecture of his course "Inferenzmethoden" at TU Darmstadt, because he was out of town. The intent of this prover was to show students a compact Prolog implementation of the (classical) *connection calculus* [2, 3] which was the topic of the lesson at that time. Slightly simplifying that code and adding the *positive start clause* technique resulted in leanCoP 1.0, whose seven lines of Prolog code is shown in the abstract of the 2003 leanCoP article. Since then, several connection provers based on or inspired by leanCoP have been developed by the author as well as by many other researchers.

This paper provides a comprehensive overview of the leanCoP connection provers and all its siblings that have been developed so far. The leanCoP provers for *classical* first-order logic (Section 2) are the foundation for all subsequent provers. ileanCoP (Section 3) and MleanCoP (Section 4) are versions of leanCoP for first-order *intuitionistic* and *modal* logics, respectively; they use *prefixes* to encode the *Kripke semantics* of these non-classical logics. nanoCoP is a generalization of leanCoP and works on formulae in non-clausal form (Section 5). The paper also presents an overview of implementations based on or inspired by leanCoP (Section 6) before concluding with a brief summary (Section 7).

AReCCa 2023: Automated Reasoning with Connection Calculi, 18 September 2023, Prague, Czech Republic
jeotten@ifi.uio.no (J. Otten)
http://jens-otten.de/ (J. Otten)
0000-0002-4331-8698 (J. Otten)

```
prove(M,I) :- append(Q,[C|R],M), \+member(-_,C),
 append(Q,R,S), prove([!],[[-!|C]|S],[],I).
prove([],_,_,_).
prove([L|C],M,P,I) :- (-N=L; -L=N) -> (member(N,P);
 append(Q,[D|R],M), copy_term(D,E), append(A,[N|B],E),
 append(A,B,F), (D==E -> append(R,Q,S); length(P,K), K<I,
 append(R,[D|Q],S)), prove(F,S,[L|P],I)), prove(C,M,P,I).
```

Figure 1: The minimal source code of the leanCoP 1.0 prover for classical first-order logic.

2. leanCoP — Classical Logic

The most popular versions of leanCoP are those for classical logic. As already mentioned, leanCoP 1.0 is the first and most basic variant of the leanCoP provers. leanCoP 2.0 contains a few selected optimizations, which significantly improve performance. leanCoP 2.1 and leanCoP 2.2 include a few additional features such as the output of a readable connection proof.

2.1. leanCoP 1.0

The minimal Prolog source code of the leanCoP 1.0 prover is shown in Fig. 1.[1] It only uses standard (built-in) Prolog predicates, such as `append(List1,List2,List3)` (`List3` is the concatenation of `List1` and `List2`), `member(Elem,List)` (is true if `Elem` is an element of `List`), `length(List,Length)` (is true if `Length` is the number of elements in `List`) and `copy_term(Term1,Term2)` (which creates a copy of `Term1` with renamed/fresh variables and unifies it to `Term2`). With a size of 333 bytes, the core code is smaller than that of the first popular *lean* prover lean*TAP* [4], whose compact Prolog code is 360 bytes in size.[2]

The prover is invoked by calling the predicate `prove(M,I)`, in which `M` is a matrix and `I` is the proof depth limit. The *matrix* M represents the input formula as a list of clauses, where each *clause* is a list of literals. Prolog terms and variables are used to represent atomic formulae and term variables, respectively, "-" is used for the negation "¬". The *proof depth limit* `I` is a natural number. The predicate succeeds iff (if and only if) there is a (connection) proof for the matrix `M`, in which the proof depth (i.e. the size of the active path) is smaller than `I`.

Example 1. $F_1 = (\,\mathrm{man}(\mathrm{Plato}) \,\wedge\, \forall x\,(\,\mathrm{man}(x) \Rightarrow \mathrm{mortal}(x)\,)\,) \,\Rightarrow\, \mathrm{mortal}(\mathrm{Plato})$ *is a first-order formula (with term variable* x*), which is equivalent to the following formula in* disjunctive normal form $\exists x(\,\neg\mathrm{man}(\mathrm{Plato}) \vee (\mathrm{man}(x) \wedge \neg\mathrm{mortal}(x)) \vee \mathrm{mortal}(\mathrm{Plato})\,)$*, which is represented by the matrix* $M_1 = \{\{\neg\mathrm{man}(\mathrm{Plato})\}, \{\mathrm{man}(x), \neg\mathrm{mortal}(x)\}, \{\mathrm{mortal}(\mathrm{Plato})\}\}$*. When the* leanCoP 1.0 *prover is invoked with the input matrix* M_1 *with*

```
prove([[-man(plato)],[man(X),-mortal(X)],[mortal(plato)]],2).
```

it returns "yes". Therefore there is a proof for M_1 *with depth smaller than 2 and the matrix* M_1 *as well as the original formula* F_1 *are* valid *(in classical logic).*

[1] Sound Prolog unification has to be used. In most Prolog systems, such as ECLiPSe or SWI Prolog, this is switched on by "`set_flag(occur_check,on).`".

[2] Note that the leanCoP code (and its calculus) is entirely different from the lean*TAP* code (and its calculus).

Axiom (A)	$\dfrac{}{\{\}, M, Path}$	
Start (S)	$\dfrac{C_2, M, \{\}}{\varepsilon,\ M,\ \varepsilon}$	C_2 is a copy of $C_1 \in M$
Reduction (R)	$\dfrac{C, M, Path \cup \{L_2\}}{C \cup \{L_1\}, M, Path \cup \{L_2\}}$	$\{L_1, L_2\}$ is σ-complementary
Extension (E)	$\dfrac{C_2 \setminus \{L_2\}, M, Path \cup \{L_1\} \quad C, M, Path}{C \cup \{L_1\}, M, Path}$	C_2 is a copy of $C_1 \in M$, $L_2 \in C_2$, $\{L_1, L_2\}$ is σ-complementary

Figure 2: The clausal connection calculus for classical logic.

2.2. The Classical Connection Calculus

leanCoP 1.0 implements the basic (clausal) connection calculus for classical logic [2, 3, 5]. In contrast to sequent calculi [6] or tableau calculi [7, 8], connection calculi are *connection-driven* leading to a *goal-oriented*, more efficient proof search. A *connection* is a set $\{A_1, \neg A_2\}$ of *literals* with the same atomic formula, but different *polarities*. It corresponds to an axiom in the sequent calculus or a closed branch in the tableau calculus. For first-order logic, a *term substitution* σ_T assigns terms to variables. A connection is σ_T*-complementary* iff $\sigma_T(A_1) = \sigma_T(A_2)$.

Definition 1 (Clausal Connection Calculus for Classical Logic). *The formal description of the (clausal)* connection calculus *for classical logic is given in Fig. 2. It consists of one axiom and three rules.*[3] *The words of the calculus are tuples "*$C, M, Path$*", where* M *is a matrix,* C *is the* (subgoal) clause *(or* ε*) and the* (active) $Path$ *is a set of literals (or* ε*). A* copy *of a clause* C *is made by renaming all variables in* C*. A* connection proof *for* M *is a derivation of* $\varepsilon, M, \varepsilon$ *with a term substitution* $\sigma = \sigma_T$ *in the connection calculus, in which all leaves are axioms.*

The rules of the calculus are applied in an *analytic* way, i.e. from bottom to top.[4] The *rigid* term substitution σ_T is calculated by a *term unification* algorithm (see, e.g. [9]) whenever a reduction or extension rule is applied. The connection calculus is sound and complete [3].

Example 2. *A formal connection proof for the matrix* M_1 *from Example 1 is given below. It is* $\sigma_T(x') = Plato$ *where the variable* x' *occurs in the (implicit) copy of the second clause of* M_1*.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\}, M_1, \{\mathit{mortal(Plato)},\ \mathit{man}(x')\}}\,A \qquad \dfrac{}{\{\}, M_1, \{\mathit{mortal(Plato)}\}}\,A}{\{\mathbf{man}(x')\}, \{\{\neg\mathbf{man(Plato)}\}, \ldots\}, \{\mathit{mortal(Plato)}\}}\,E \qquad \dfrac{}{\{\}, M_1, \{\}}\,A}{\{\mathbf{mortal(Plato)}\}, \{\{\neg \mathit{man(Plato)}\}, \{\mathit{man}(x), \neg\mathbf{mortal}(x)\}, \{\mathit{mortal(Plato)}\}\}, \{\}}\,E}{\varepsilon, \{\{\neg \mathit{man(Plato)}\}, \{\mathit{man}(x), \neg \mathit{mortal}(x)\}, \{\mathit{mortal(Plato)}\}\}, \varepsilon}\,S$$

[3] This formalization of the connection calculus was first published in [1] together with the code of leanCoP 1.0.

[4] During the proof search, *backtracking* might be necessary (in case the chosen rule or rule instance does not lead to a proof) when more than one rule is applicable or for alternative clauses C_1 or literals L_2 (in the start, reduction or extension rule). No backtracking is necessary for choosing the literal L_1 (in the reduction or extension rule).

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

Figure 3: The minimal source code of the leanCoP 2.0 prover for classical first-order logic.

2.3. leanCoP 2.0

Even though leanCoP 1.0 already outperformed the famous Otter theorem prover [10] on a small number of problems [1], the development of the next version took a big step forward. In the year 2006, the problems of the *MPTP challenge* [11] motivated the integration of further optimizations into leanCoP: regularity, lemmata, restricted backtracking, conjecture start clauses, definitional clausal form, strategy scheduling and a more efficient representation of the input clauses. It resulted in leanCoP 2.0, whose minimal Prolog code of the core prover is shown in Fig. 3.[5] This minimal version of leanCoP 2.0 is still only 555 bytes in size.

A few basic but effective techniques were carefully selected and integrated into leanCoP 2.0. *Regularity* reduces the search space by ensuring that no literal occurs more than once on the active path [5]. *Lemmata* (or *factorization*) reuses the subproof of a literal in order to solve the same literal at a later point in the proof search [5]. *Restricted backtracking* is an effective (but incomplete) technique to prune the search space in the (non-confluent) connection calculus [12]. It cuts off any alternative applications of reduction and extensions rules once a literal from the subgoal clause has been solved.[6] Furthermore, backtracking can also be restricted when selecting the clause C_1 in the start rule by omitting alternative start clauses. *Conjecture start clauses* restrict the start rule for formulae of the form $(A_1 \wedge \ldots \wedge A_n) \Rightarrow C$ to clauses of the conjecture C instead of the (default) positive clauses. A *definitional clausal form* translation, which is done in a preprocessing step, introduces definitions for certain subformulae [12], hereby reducing the size of the resulting matrix. *Reordering clauses* is done in a preprocessing step and modifies (indirectly) the proof search order, by reordering the clauses in the input matrix. *Strategy scheduling* uses a sequence of different strategies to prove a formula, which are specified in the last argument of the `prove` predicate [13, 12]. For example, the first strategy "`[cut,comp(7)]`" used by leanCoP 2.0 uses restricted backtracking and restarts with a complete proof search (without restricted backtracking) if a proof depth of 7 is reached [12].

[5] The source code shown in Fig. 3 uses only standard (built-in) Prolog predicates and implements the whole proof search. The input matrix is stored in Prolog's database using the `lit` predicate (see below for details).

[6] The literal L_1 in Fig. 2 is called *solved*; in case of the extension rule, there also has to be a proof for the left premise.

Using a *lean Prolog technology* [12], the clauses of the input matrix M are stored in Prolog's database in a preprocessing step. For every clause $C \in M$ and for all literals $\texttt{L} \in C$ the fact "`lit(L,C1,Grnd).`" is stored, where `C1`$=C\backslash\{$`L`$\}$ and `Grnd` is `g` if C is ground (does not contain any variables) and `n`, otherwise. This integrates the main advantage of the "Prolog technology" approach [14] into leanCoP, using Prolog's fast indexing mechanism to find connections.

Fig. 4 (ignoring the underlined code) shows the *comprehensive* source code of leanCoP 2.0. The core prover is invoked with `prove(1,Set)`, where `Set` is a strategy and the initial limit for the proof depth (size of the active path) is 1. This predicate succeeds iff there is a connection proof for the matrix/clauses stored in Prolog's database. The proof search starts by applying the start rule (lines a–c). The path limit is used to perform iterative deepening on the size of the active path (lines e–h), which is necessary for completeness. Afterwards, the reduction rule (lines 2, 5, 8, 17) and the extension rule (lines 2, 5, 11–14, 17) are repeatedly applied, until the branch in the derivation can be closed by an axiom (line 1). A *controlled iterative deepening* stops the proof search if the current path limit for the size of the active path is not exceeded (line g and 13). This yields a decision procedure for ground (e.g. propositional) formulae and also allows for refuting some (invalid) first-order formulae. The proof search optimizations integrated into the core prover are *regularity* (line 4), *lemmata* (line 6), and *restricted backtracking* (line 16); see [12] for details. The term substitution σ_T is stored implicitly by Prolog. The additional optimizations improve the performance of leanCoP significantly, in particular for large formulae [12].

```
(a)   prove(PathLim,Set,Proof) :-
(b)       \+member(scut,Set) -> prove([-(#)],[],PathLim,[],Set,[Proof]) ;
(c)       lit(#,C,_) -> prove(C,[-(#)],PathLim,[],Set,Proof1),
(d)       Proof=[C|Proof1].
(e)   prove(PathLim,Set,Proof) :-
(f)       member(comp(Limit),Set), PathLim=Limit -> prove(1,[],Proof) ;
(g)       (member(comp(_),Set);retract(pathlim)) ->
(h)       PathLim1 is PathLim+1, prove(PathLim1,Set,Proof).

(1)   prove([],_,_,_,_,[]).
(2)   prove([Lit|Cla],Path,PathLim,Lem,Set,Proof) :-
(3)       Proof=[[[NegLit|Cla1]|Proof1]|Proof2],
(4)       \+ (member(LitC,[Lit|Cla]), member(LitP,Path), LitC==LitP),
(5)       (-NegLit=Lit;-Lit=NegLit) ->
(6)          ( member(LitL,Lem), Lit==LitL, Cla1=[], Proof1=[]
(7)            ;
(8)            member(NegL,Path), unify_with_occurs_check(NegL,NegLit),
(9)            Cla1=[], Proof1=[]
(10)           ;
(11)           lit(NegLit,Cla1,Grnd1),
(12)           ( Grnd1=g -> true ; length(Path,K), K<PathLim -> true ;
(13)             \+ pathlim -> assert(pathlim), fail ),
(14)           prove(Cla1,[Lit|Path],PathLim,Lem,Set,Proof1)
(15)         ),
(16)         ( member(cut,Set) -> ! ; true ),
(17)         prove(Cla,Path,PathLim,[Lit|Lem],Set,Proof2).
```

Figure 4: The source code of the leanCoP 2.0 and leanCoP 2.1/2.2 provers for classical first-order logic.

Table 1
leanCoP 2.0 and ileanCoP 1.2 at CASC 2007.

Prover	proved	refuted	rating >0.68	CASC package size
Vampire 9.0	270	-	59	7700 kB
E 0.999	248	-	41	17800 kB
iProver 0.2	201	-	18	5800 kB
Equinox 1.2	173	-	19	6200 kB
leanCoP 2.0	**160**	-	**21**	**26 kB**
Otter 3.3	138	-	6	3200 kB
Metis 2.0	117	-	2	13700 kB
Geo 2007f	104	-	5	6400 kB
ileancop 1.2	**103**	9	**13**	**27 kB**
Muscadet 2.7	37	-	11	3000 kB

At CASC in 2007, leanCoP 2.0 [13, 12] proved more problems (out of 300 problems with a time limit of 360 seconds) than four other provers in the main FOF division [15], including 21 "difficult" problems; see Table 1. It proved four problems not solved by the Vampire prover [16], won both the bushy and chainy 100$ MPTP challenges, and was awarded *Best Newcomer* [15].

Example 3. *The preprocessing step of* leanCoP 2.0 *stores the clauses of matrix M_1 from Example 1, $M_1 = \{\{\neg\text{man}(\text{Plato})\}, \{\text{man}(x), \neg\text{mortal}(x)\}, \{\text{mortal}(\text{Plato})\}\}$, in the following form:*[7]

```
lit(#,[mortal(plato)],g).   (clause 3)    lit(man(X),[-mortal(X)],n).   (clause 2)
lit(mortal(plato),[#],g).   (clause 3)    lit(-mortal(X),[man(X)],n).   (clause 2)
lit(-man(plato),[],g).      (clause 1)
```

Then when the leanCoP 2.0 *prover is invoked with "*`prove(1,[cut,comp(7)]).`*" it returns "yes". Therefore, the matrix M_1 as well as the original formula F_1 are* valid *(in classical logic).*

2.4. leanCoP 2.1/2.2

In leanCoP 2.1 a few additional features have been added to leanCoP 2.0. leanCoP 2.1 directly supports the TPTP input syntax (for FOF) with equality, where equality axioms are automatically added if necessary. The fixed strategy scheduling is now controlled by a shell script, and support for SWI and SICStus Prolog is added. Furthermore, an additional `Proof` argument is added to the core prover, which returns a compact connection proof. Afterwards, this proof can be translated into different formats: `connect` (standard connection proof), `leantptp` (proof in a TPTP-like syntax) and `readable` (readable proof in natural language). In leanCoP 2.2 the (detailed) proof output has been improved and the fixed strategy scheduling optimized.

The source code of the leanCoP 2.1/2.2 core prover is shown in Fig. 4. The underlined code, which collects the proof, was added to leanCoP 2.0, nothing else was changed. The core prover is invoked with `prove(1,Set,Proof)`, where `Proof` returns the found connection proof.

At CASC in 2009, the leanCoP 2.1 prover, and at CASC in 2010, the leanCoP 2.2 prover were among the top three provers that return a proof in the main FOF division.[8]

[7] The special literal `#` is added to all positive clauses and the proof starts with the subgoal `[-(#)]` (line b/c in Fig. 4).
[8] See https://www.tptp.org/CASC/J5/ and https://www.tptp.org/CASC/22/

Example 4. *For the matrix M_1 stored as shown in Example 3, the* leanCoP 2.1/2.2 *core prover invoked with "*`prove(1,[cut,comp(7)],Proof).`*" returns the* compact *connection proof*

```
Proof = [[#,mortal(plato)],[[-(mortal(plato)),man(plato)],[[-(man(plato))]]]]
```

which is a nested list of clauses C_2 used in start/extension steps (or literals L_2 in reduction steps).

3. ileanCoP — Intuitionistic Logic

Intuitionistic logic [17] has applications, e.g., in program synthesis and within interactive proof assistants such as Coq [18] and NuPRL [19] used for constructing provably correct software. Intuitionistic and classical logic share the same *syntax*, but their *semantics* is different. For example, the formula *man(Socrates)* $\vee$ $\neg$*man(Socrates)* is valid in classical logic, but not in intuitionistic logic. The semantics of intuitionistic logic requires a proof for *man(Socrates)* or for $\neg$*man(Socrates)*, which do not exist. Every formula that is valid in intuitionistic logic is also valid in classical logic, but not vice versa. Gentzen already provided a sequent calculus LJ for intuitionistic first-order logic [6]. Wallen extended Bibel's *matrix characterization* [3] to intuitionistic logic by using prefixes to encode the *Kripke semantics* [20]. This characterization, originally formulated for formulae in non-clausal form, was adapted to clausal form by using an extended Skolemization [21]. It serves as the basis for a clausal connection calculus for intuitionistic first-order logic and the ileanCoP implementation [21].

3.1. The Intuitionistic Connection Calculus

For *intuitionistic* logic the matrix and the calculus are extended by prefixes, representing world paths in the Kripke semantics; see [20, 22]. A *prefix* p is a string consisting of variables (denoted by V) and constants (denoted by a) and assigned to each literal (and subformula).

Definition 2. *The* intuitionistic matrix $M(F^{pol}{:}p)$ of a prefixed formula $F^{pol}{:}p$ *with polarity* $pol{\in}\{0,1\}$ *(see [7, 20]) is defined according to Table 2.* $M_G \cup_\beta M_H := \{C_G \cup C_H \mid C_G \in M_G, C_H \in M_H\}$, x^* *is a new term variable,* t^* *is the Skolem term* $f^*(x_1,\ldots,x_n)$, V^* *is a new prefix variable,* a^* *is the prefix constant* $f^*(x_1,\ldots,x_n)$, f^* *is a new function symbol and* $x_1,\ldots,x_n$ *are all free term and prefix variables in* $A^0:p$, $(\neg G)^0:p$, $(G \Rightarrow H)^0:p$, $(\forall x G)^0:p$ *or* $(\exists x G)^1:p$, *respectively. The* intuitionistic matrix $M^i(F)$ *of a formula* F *is the intuitionistic matrix* $M(F^0:\varepsilon)$.

Table 2
The prefixed (clausal) matrix for intuitionistic logic.

	$F^{pol}:p$	$M(F^{pol}:p)$
	$A^0:p$	$\{\{A^0:p\underline{a^*}\}\}$
	$A^1:p$	$\{\{A^1:p\underline{V^*}\}\}$
α	$(\neg G)^0:p$	$M(G^1:p\underline{a^*})$
	$(\neg G)^1:p$	$M(G^0:p\underline{V^*})$
	$(G\wedge H)^1:p$	$M(G^1:p) \cup M(H^1:p)$
	$(G\vee H)^0:p$	$M(G^0:p) \cup M(H^0:p)$
	$(G\Rightarrow H)^0:p$	$M(G^1{:}p\underline{a^*}) \cup M(H^0{:}p\underline{a^*})$

	$F^{pol}:p$	$M(F^{pol}:p)$
β	$(G\wedge H)^0:p$	$M(G^0:p) \cup_\beta M(H^0:p)$
	$(G\vee H)^1:p$	$M(G^1:p) \cup_\beta M(H^1:p)$
	$(G\Rightarrow H)^1:p$	$M(G^0{:}p\underline{V^*}) \cup_\beta M(H^1{:}p\underline{V^*})$
γ	$(\forall x G)^1:p$	$M(G[x\backslash x^*]^1:p\underline{V^*})$
	$(\exists x G)^0:p$	$M(G[x\backslash x^*]^0:p)$
δ	$(\forall x G)^0:p$	$M(G[x\backslash t^*]^0:p\underline{a^*})$
	$(\exists x G)^1:p$	$M(G[x\backslash t^*]^1:p)$

Axiom (A)	$\dfrac{}{\{\}, M, Path}$	*Start (S)*	$\dfrac{C_2, M, \{\}}{\varepsilon,\ M,\ \varepsilon}$	C_2 is a copy of $C_1 \in M$
Reduction (R)	$\dfrac{C, M, Path \cup \{L_2 : p_2\}}{C \cup \{L_1 : p_1\}, M, Path \cup \{L_2 : p_2\}}$		$\{L_1 : p_1, L_2 : p_2\}$ is σ-complementary	
Extension (E)	$\dfrac{C_2 \setminus \{L_2 : p_2\}, M, Path \cup \{L_1 : p_1\} \quad C, M, Path}{C \cup \{L_1 : p_1\}, M, Path}$			C_2 is a copy of $C_1 \in M$, $L_2 : p_2 \in C_2$, $\{L_1 : p_1, L_2 : p_2\}$ is σ-complementary

Figure 5: The clausal connection calculus for intuitionistic and modal logic.

A *prefix substitution* σ_P assigns strings to prefix variables and is calculated by a *prefix unification* [21]. In intuitionistic logic, a connection $\{A_1^0 : p_1, A_2^1 : p_2\}$ is *σ-complementary* iff $\sigma_T(A_1) = \sigma_T(A_2)$ and $\sigma_P(p_1) = \sigma_P(p_2)$ for a combined substitution $\sigma = (\sigma_T, \sigma_P)$.

Definition 3 (Clausal Connection Calculus for Intuitionistic Logic). *The (clausal)* connection calculus *for* intuitionistic logic *[21] is given in Fig. 5. An* intuitionistic connection proof *for F is a derivation of $\varepsilon, M^i(F), \varepsilon$ with an* admissible *(see [20, 21]) combined substitution $\sigma = (\sigma_T, \sigma_P)$, in which all leaves are axioms.*

Example 5. *The intuitionistic matrix of the formula F_1 from Example 1 is $M_1^i = M^i(F_1) =$ $\{\{\text{man(Plato)}^1 : a_1 V_1\},\ \{\text{man}(x)^0 : a_1 V_2\ a_2(x),\ \text{mortal}(x)^1 : a_1 V_2 V_3\},\ \{\text{mortal(Plato)}^0 : a_1 a_3\}\}$. The intuitionistic connection proof for M_1^i uses the same steps as the classical proof in Example 2, but with a combined substitution $\sigma = (\sigma_T, \sigma_P)$ with $\sigma_T(x') = \text{Plato}$, $\sigma_P(V_1) = a_2(\text{Plato})$, $\sigma_J(V_2) = \varepsilon$ and $\sigma_P(V_3) = a_3$ (where ε is the empty string). Therefore, M_1^i and F_1 are valid in intuitionistic logic. The intuitionistic matrix of the formula $F_2 = \text{man(Socrates)} \vee \neg\, \text{man(Socrates)}$ is $M_2^i = M^i(F_2) = \{\{\text{man(Socrates)}^0 : a_1\}, \{\text{man(Socrates)}^1 : a_2\}\}$. It contains the only connection $\{\text{man(Socrates)}^0 : a_1, \text{man(Socrates)}^1 : a_2\}$, which cannot be complementary as there is no σ_P that unifies the two prefixes a_1 and a_2. Therefore, M_2^i and F_2 are* not *valid in intuitionistic logic.*

3.2. ileanCoP 1.0/1.2

ileanCoP is a compact Prolog implementation of the clausal connection calculus for intuitionistic logic and extends the classical connection prover leanCoP by (a) *prefixes* (that are added to the literals in the matrix in a preprocessing step), (b) *a set of prefix equations* (that are collected during the proof search), (c) *a set of term variables* (with their prefixes) assigned to each clause (in order to check the admissibility of σ, also called *domain condition*), and (d) a *prefix unification algorithm* (that unifies the prefixes of the literals in each connection).

ileanCoP 1.0 [21] is based on (classical) leanCoP 1.0 and implements the basic intuitionistic connection calculus. ileanCoP 1.2 [13] integrates all the additional optimization techniques and strategies of leanCoP 2.0. The minimal source code of the ileanCoP 1.2 core prover is shown in Fig. 6. The underlined code was added to the classical prover leanCoP 2.0 shown in Fig. 3; no further changes were made. In a preprocessing step the clauses of the intuitionistic matrix are written into Prolog's database using the `lit` predicate (see also Section 2.3).

```
prove(I,S) :- ( \+member(scut,S) ->
    prove([(-(#)):(-[])],[],I,[],[Z,T],S) ;
    lit((#):_,G:C,_) -> prove(C,[(-(#)):(-[])],I,[],[Z,R],S),
    append(R,G,T) ), domain_cond(T), prefix_unify(Z).
prove(I,S) :- member(comp(L),S), I=L -> prove(1,[]) ;
    (member(comp(_),S);retract(p)) -> J is I+1, prove(J,S).
prove([],_,_,_,[[],[]],_).
prove([L:U|C],P,I,Q,[Z,T],S) :- \+ (member(A,[L:U|C]), member(B,P),
    A==B), (-N=L;-L=N) -> ( member(D,Q), L:U==D, X=[], O=[] ;
    member(E:V,P), unify_with_occurs_check(E,N),
    \+ \+ prefix_unify([U=V]), X=[U=V], O=[] ;
    lit(N:V,M:F,H), \+ \+ prefix_unify([U=V]),
    (H=g -> true ; length(P,K), K<I -> true ;
    \+p -> assert(p), fail), prove(F,[L:U|P],I,Q,[W,R],S),
    X=[U=V|W], append(R,M,O) ), (member(cut,S) -> ! ; true),
    prove(C,P,I,[L:U|Q],[Y,J],S), append(X,Y,Z), append(J,O,T).
```

Figure 6: The source code of the ileanCoP 1.2/MleanCoP 1.2 core provers for intuitionistic/modal logic.

The prover is invoked with `prove(1,S)`, where `S` is a strategy (see Section 2.3) and the start limit for the size of the active path is 1. The predicate succeeds if there is an intuitionistic connection proof for the stored input clauses. First, ileanCoP performs a classical proof search. After a classical proof is found, the domain condition is checked and the prefixes of the literals in each connection are unified [21]. This is done by the predicates `domain_cond` and `prefix_unify`, respectively, in the fourth line of the code in Fig. 6. These are the only additional predicates invoked during the proof search and are implemented by 5 and 21 lines of Prolog code, respectively. The substitutions σ_T and σ_P are stored implicitly by Prolog.

At CASC in 2007, see Table 1, ileanCoP 1.2 [13] proved two problems intuitionistically that Vampire [16] did not prove classically even though reasoning in intuitionistic logic is significantly harder (for propositional logic it is PSPACE-complete [23] compared to NP-complete [24]). For more than 10 years ileanCoP 1.2 was the fastest prover for intuitionistic first-order logic.

4. MleanCoP — **Modal Logic**

Modal logics [25, 26] are among the most popular *non-classical* logics and have applications in, e.g., planning, natural language processing, and program verification. Modal logics extend the language of classical logic with the unary *modal operators* $\Box$ and $\Diamond$ representing the modalities "it is necessarily true that" and "it is possibly true that", respectively. For example, the proposition "if Plato is necessarily a man, then Plato is possibly a man" can be represented by the modal formula $\Box\, man(Plato) \Rightarrow \Diamond\, man(Plato)$.

The *Kripke semantics* of the (standard) modal logics is defined by a set of *worlds* and a binary *accessibility relation* between these worlds. In each single world the classical semantics applies to the classical connectives, whereas the modal operators $\Box$ and $\Diamond$ are interpreted with respect to accessible worlds. There exist a broad range of different modal logics and the properties of the accessibility relation specify the particular modal logic, such as D, T, S4 or S5.

Table 3
The prefixed (clausal) matrix for modal logic.

	$F^{pol} : p$	$M(F^{pol} : p)$
	$A^{pol} : p$	$\{\{A^{pol} : p\}\}$
α	$(\neg G)^{pol} : p$	$M(G^{1-pol} : p)$
	$(G \wedge H)^1 : p$	$M(G^1 : p) \cup M(H^1 : p)$
	$(G \vee H)^0 : p$	$M(G^0 : p) \cup M(H^0 : p)$
	$(G \Rightarrow H)^0 : p$	$M(G^1 : p) \cup M(H^0 : p)$
β	$(G \wedge H)^0 : p$	$M(G^0 : p) \cup_\beta M(H^0 : p)$
	$(G \vee H)^1 : p$	$M(G^1 : p) \cup_\beta M(H^1 : p)$
	$(G \Rightarrow H)^1 : p$	$M(G^0 : p) \cup_\beta M(H^1 : p)$

	$F^{pol} : p$	$M(F^{pol} : p)$
γ	$(\forall x G)^1 : p$	$M(G[x \backslash x^*]^1 : p)$
	$(\exists x G)^0 : p$	$M(G[x \backslash x^*]^0 : p)$
δ	$(\forall x G)^0 : p$	$M(G[x \backslash t^*]^0 : p)$
	$(\exists x G)^1 : p$	$M(G[x \backslash t^*]^1 : p)$
ν	$(\Box G)^1 : p$	$M(G^1 : p\underline{V^*})$
	$(\Diamond G)^0 : p$	$M(G^0 : p\underline{V^*})$
π	$(\Box G)^0 : p$	$M(G^0 : p\underline{a^*})$
	$(\Diamond G)^1 : p$	$M(G^1 : p\underline{a^*})$

Proof calculi for modal logic often use prefixes [27] and so does Wallen's matrix characterizations [20]. Again, this characterization was adapted and serves as the basis for the clausal connection calculus and the MleanCoP implementation for several modal logics [28, 29].[9]

4.1. The Modal Connection Calculus

For *modal* logic the matrix and the calculus are extended by prefixes, representing world paths in the Kripke semantics; see [20, 22]. Again, a *prefix* p is a string consisting of variables (denoted by V) and constants (denoted by a) and assigned to each literal (and subformula).

Definition 4. *The* modal matrix $M(F^{pol}{:}p)$ of a prefixed formula $F^{pol}{:}p$ *with polarity* $pol \in \{0,1\}$ *is defined according to Table 3.* $M_G \cup_\beta M_H := \{C_G \cup C_H \mid C_G \in M_G, C_H \in M_H\}$, x^* *is a new term variable,* t^* *is the Skolem term* $f^*(x_1, \ldots, x_n)$, V^* *is a new prefix variable,* a^* *is the prefix constant* $f^*(x_1, \ldots, x_n)$, f^* *is a new function symbol and* $x_1, \ldots, x_n$ *are all free term and prefix variables in* $(\forall x G)^0 : p$, $(\exists x G)^1 : p$, $(\Box G)^0 : p$ *or* $(\Diamond G)^1 : p$, *respectively. The* modal matrix $M^m(F)$ *of a formula* F *is the modal matrix* $M(F^0 : \varepsilon)$.

A *prefix substitution* σ_P assigns strings to prefix variables and is calculated by a *prefix unification* [28, 31, 32]. In modal logic, a connection $\{A_1^0 : p_1, A_2^1 : p_2\}$ is σ*-complementary* iff $\sigma_T(A_1) = \sigma_T(A_2)$ and $\sigma_P(p_1) = \sigma_P(p_2)$ for a combined substitution $\sigma = (\sigma_T, \sigma_P)$. The prefix unification procedure depends on the specific modal logic and its domain condition and has to respect its *accessibility condition*: $|\sigma_P(V)| = 1$ for the modal logic D and $|\sigma_P(V)| \leq 1$ for the modal logic T (for all prefix variables V); for S5 only the last character (or ε if the prefix is empty) of each prefix has to be unified; there is no restriction for the modal logic S4 [28, 29].

Definition 5 (Clausal Connection Calculus for Modal Logic). *The (clausal)* connection calculus *for* modal logic *[28, 31] is given in Fig. 5. A* modal connection proof *for the modal formula* F *is a derivation of* $\varepsilon, M^m(F), \varepsilon$ *with an* admissible *(see [20, 28]) combined substitution* $\sigma = (\sigma_T, \sigma_P)$, *in which all leaves are axioms.*

[9] The calculus and the implementation for modal logic are very similar to the ones for intuitionistic logic in Section 3; only the specification of prefixes, the prefix unification and the domain condition differ. Indeed, Gödel showed that intuitionistic logic can be embedded into the modal logic S4 with cumulative domains [30]. For historical reasons and to keep the notation simple for different audiences, these logics are covered in separate sections.

Example 6. *The modal matrix of* $F_3 = \Box\,\mathrm{man} \Rightarrow \Box\Diamond\,\mathrm{man}$ *is* $M_3^m = M^m(F_3) = \{\{\mathrm{man}^1 : V_1\}, \{\mathrm{man}^0 : a_1 V_2\}\}$. *The modal connection proof for* M_3^m *has one connection* $\{\mathrm{man}^1 : V_1,\ \mathrm{man}^0 : a_1 V_2\}$, *which is* σ*-complementary for the combined (admissible) substitutions* $\sigma_P(V_1) = a_1$, $\sigma_P(V_2) = \varepsilon$ *(empty string) for T,* $\sigma_P(V_1) = a_1 V_2$ *for S4, and* $\sigma_P(V_1) = V_2$ *for S5. Therefore,* M_3^m *and* F_3 *are valid in the modal logics T, S4 and S5, but* not *in the modal logic D (which requires* $|\sigma_P(V_i)| = 1$*).*

4.2. MleanCoP 1.2/1.3

MleanCoP is a compact implementation of the connection calculus for the modal logics D, T, S4 and S5 with constant, cumulative and varying domains. MleanCoP 1.2 [28] is based on leanCoP 2.0 extended by prefixes and a prefix unification algorithm used to calculate the prefix substitution σ_P. The minimal source code of the MleanCoP 1.2 core prover is shown in Fig. 6. The underlined code was added to the classical prover, no further changes were made. In a preprocessing step the clauses of the modal matrix are written into Prolog's database using the `lit` predicate (see also Section 2.3). The prover is invoked with `prove(1,S)`, which succeeds if there is a modal connection proof for the stored input clauses. After a classical proof is found, the domain condition is checked (`domain_cond`) and the prefixes of the literals in each connection are unified (`prefix_unify`), which depend on the specific modal logic [20, 28].

MleanCoP 1.3 [29] includes the following improvements: support for heterogenous multimodal logics, output of a compact modal connection proof, support for the modal QMLTP input syntax [33] and an improved strategy scheduling. Up to the release of nanoCoP-M 2.0, MleanCoP was the fastest prover for the first-order modal logics D, T, S4 and S5 [34, 35].

5. nanoCoP — Non-Clausal Reasoning

nanoCoP [31, 36, 37, 35] is a series of compact Prolog implementations of the non-clausal connection calculus. The *non-clausal* connection calculus for classical [38, 39] and non-classical logics [37] generalizes the *clausal* connection calculus [1, 2, 3].

5.1. The Non-Clausal Connection Calculus

In the *clausal* connection calculus a matrix is a set of clauses, where a clause is a set of literals. The *non-clausal* connection calculus works on *non-clausal* matrices, in which a matrix is a set of clauses and a clause is a set of literals *and* (sub)matrices. It can be seen as a representation of a formula in negation normal form. For the non-classical logics, the non-clausal matrix and calculus are extended by prefixes p, i.e., strings consisting of variables (V) and constants (a).

Definition 6. *The* classical non-clausal matrix $M(F^{pol})$ *of* F^{pol} *with polarity* $pol \in \{0, 1\}$ *is defined (inductively) according to Table 4; the prefixes* $: p \ldots$ *and the last two lines are to be ignored. The* non-classical non-clausal matrix $M^{i/m}(F^{pol}{:}p)$ *of* $F^{pol}{:}p$ *with polarity* $pol \in \{0, 1\}$ *is defined (inductively) according to Table 4; for intuitionistic logic (*M^i*) the last two lines are to be ignored, for modal logic (*M^m*) the underlined characters are to be ignored).* x^* *is a new term variable,* t^* *is the Skolem term* $f^*(x_1, \ldots, x_n)$, V^* *is a new prefix variable,* a^* *is the prefix constant* $f^*(x_1, \ldots, x_n)$, f^* *is a new function symbol and* $x_1, \ldots, x_n$ *are all free term and prefix variables in the corresponding formula* $F^{pol} : p$. *The* classical non-clausal matrix $M(F)$ *of* F *is the matrix* $M(F^0)$. *The* non-classical non-clausal matrix $M^{i/m}(F)$ *of* F *is the matrix* $M^{i/m}(F^0 : \varepsilon)$.

Table 4
The (prefixed) non-clausal matrix for classical, intuitionistic and modal logic.

type	$F^{pol}:p$	$M(F^{pol}:p)$
atom	$A^0:p$	$\{\{A^0:p\underline{a^*}\}\}$
α	$(G\wedge H)^1:p$	$\{\{M(G^1:p)\}\},\{\{M(H^1:p)\}\}$
	$(G\vee H)^0:p$	$\{\{M(G^0:p)\}\},\{\{M(H^0:p)\}\}$
	$(G\Rightarrow H)^0{:}p$	$\{\{M(G^1:p\underline{a^*})\}\},\{\{M(H^0:p\underline{a^*})\}\}$
β	$(G\wedge H)^0:p$	$\{\{M(G^0:p),M(H^0:p)\}\}$
	$(G\vee H)^1:p$	$\{\{M(G^1:p),M(H^1:p)\}\}$
	$(G\Rightarrow H)^1{:}p$	$\{\{M(G^0:p\underline{V^*}),M(H^1:p\underline{V^*})\}\}$
ν	$(\Box G)^1:p$	$M(G^1:pV^*)$
	$(\Diamond G)^0:p$	$M(G^0:pV^*)$

type	$F^{pol}:p$	$M(F^{pol}:p)$
atom	$A^1:p$	$\{\{A^1:p\underline{V^*}\}\}$
α	$(\neg G)^0:p$	$M(G^1:p\underline{a^*})$
	$(\neg G)^1:p$	$M(G^0:p\underline{V^*})$
γ	$(\forall xG)^1:p$	$M(G[x\backslash x^*]^1:p\underline{V^*})$
	$(\exists xG)^0:p$	$M(G[x\backslash x^*]^0:p)$
δ	$(\forall xG)^0:p$	$M(G[x\backslash t^*]^0:p\underline{a^*})$
	$(\exists xG)^1:p$	$M(G[x\backslash t^*]^1:p)$
π	$(\Box G)^0:p$	$M(G^0:pa^*)$
	$(\Diamond G)^1:p$	$M(G^1:pa^*)$

A *term substitution* σ_T assigns terms to variables. For classical logic, a connection $\{A_1^0, A_2^1\}$ is σ_T*-complementary* iff $\sigma_T(A_1)=\sigma_T(A_2)$. For the non-classical logics, additionally, a *prefix substitution* σ_P assigns strings to prefix variables and is calculated by a *prefix unification* that depends on the specific non-classical logic (see also Sections 3.1 and 4.1). For the non-classical logics, a connection $\{A_1^0:p_1, A_2^1:p_2\}$ is σ*-complementary* iff $\sigma_T(A_1)=\sigma_T(A_2)$ and $\sigma_P(p_1)=\sigma_P(p_2)$ for a combined substitution $\sigma=(\sigma_T,\sigma_P)$.

In the non-clausal connection calculus, the extension rule is generalized to (nested) *extension clauses (e-clauses)* and a *decomposition rule* is added, which splits subgoal clauses [38, 31].

Definition 7 (Non-Clausal Connection Calculus). *The* non-clausal connection calculus *for classical [38] and non-classical [37] logic is given in Fig. 7 (for classical logic, the underlined text is to be ignored). A* classical non-clausal connection proof *for F is a derivation of $\varepsilon, M, \varepsilon$ in the non-clausal connection calculus with a term substitution $\sigma=\sigma_T$, in which all leaves are axioms. An* intuitionistic/modal connection proof *for F is a derivation of $\varepsilon, M^{i/m}(F), \varepsilon$ with an* admissible *(see [20, 37]) combined substitution $\sigma=(\sigma_T,\sigma_P)$, in which all leaves are axioms.*

Axiom (A) $\dfrac{}{\{\}, M, \mathit{Path}}$ *Start (S)* $\dfrac{C_2, M, \{\}}{\varepsilon,\ M,\ \varepsilon}$ and C_2 is copy of $C_1 \in M$

Reduction (R) $\dfrac{C, M, \mathit{Path}\cup\{L_2{:}\ \underline{p_2}\}}{C\cup\{L_1:\underline{p_1}\}, M, \mathit{Path}\cup\{L_2:\underline{p_2}\}}$ and $\{L_1{:}\underline{p_1}, L_2{:}\underline{p_2}\}$ is σ-complementary

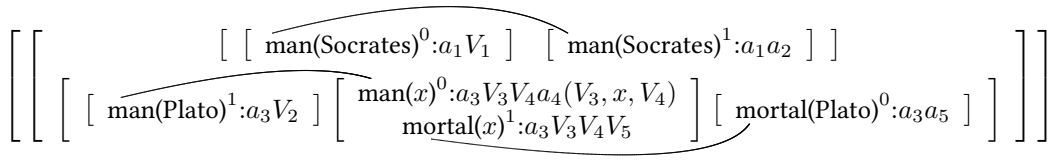
Extension (E) $\dfrac{C_3, M[C_1\backslash C_2], \mathit{Path}\cup\{L_1:\underline{p_1}\} \quad C, M, \mathit{Path}}{C\cup\{L_1{:}\ \underline{p_1}\}, M, \mathit{Path}}$

and $C_3{:=}\beta\text{-}clause_{L_2}(C_2)$, C_2 is copy of C_1, C_1 is e-clause of M wrt. $\mathit{Path}\cup\{L_1:\underline{p_1}\}$, C_2 contains $L_2:\underline{p_2}$, $\{L_1{:}\underline{p_1}, L_2{:}\underline{p_2}\}$ is σ-complementary

Decomposition (D) $\dfrac{C\cup C_1, M, \mathit{Path}}{C\cup\{M_1\}, M, \mathit{Path}}$ and $C_1 \in M_1$

Figure 7: The non-clausal connection calculus for classical, intuitionistic and modal logic.

Example 7. *The (simplified) intuitionistic non-clausal matrix of* $F_4 = (\,\text{man(Socrates)} \Rightarrow \text{man(Socrates)}\,) \wedge (\,(\,\text{man(Plato)} \wedge \forall x(\text{man}(x) \Rightarrow \text{mortal}(x))\,) \Rightarrow \text{mortal(Plato)}\,)$ *is* $M_4^i = M^i(F_4) = \{\ \{\ \{\{\text{man(Socrates)}^0 : a_1V_1\}, \{\text{man(Socrates)}^1 : a_1a_2\}\}, \{\{\text{man(Plato)}^1 : a_3V_2\}, \{\text{man}(x)^0 : a_3V_3V_4a_4(V_3, x, V_4), \text{mortal}(x)^1 : a_3V_3V_4V_5\}, \{\text{mortal(Plato)}^0 : a_3a_5\}\}\ \}\ \}$. *The graphical non-clausal connection proof for* M_4^i *with* $\sigma = (\sigma_T, \sigma_P)$ *and* $\sigma_T(x) = \text{Plato}$, $\sigma_P(V_1) = a_2$, $\sigma_P(V_2) = a_4(\varepsilon, \text{Plato}, \varepsilon)$, $\sigma_P(V_3) = \varepsilon$, $\sigma_J(V_4) = \varepsilon$, $\sigma_P(V_5) = a_5$ *is depicted below.*



5.2. nanoCoP

nanoCoP is a compact Prolog implementation of the non-clausal connection calculus for classic first-order logic (with equality) [31, 36, 35]. It is an extension of the leanCoP code (see Fig. 4).

In the first step, the input formula F is translated into a non-clausal matrix $M = M(F)$ according to Table 4 (for intuitionistic and modal logic, prefixes are added to all literals in the non-clausal matrix $M^{i/m}(F)$). Every (sub-)clause $(I, V, FV) : C$ and submatrix $J : M$ are marked with unique indices I and J, sets V of (free) term and prefix variables that are newly introduced in C (and sets FV including pairs $x : pre(x)$ of free term variables and their prefixes, necessary to check if σ is admissible for the non-classical logics). In Prolog, literals with polarity 1 are marked with "-". In the second step, for every literal `Lit` in M the fact `lit(Lit,ClaB,ClaC,Grnd)` is asserted into Prolog's database where `ClaC` $\in M$ is the (largest) clause in which `Lit` occurs, `ClaB` is β-*clause*$_{Lit}$(`ClaC`) [38] and `Grnd` is `g` iff the smallest clause in which `Lit` occurs is ground (i.e., does not contain variables).

The source code of the nanoCoP 2.0 core prover is shown in Fig. 8. The underlined code is necessary only for the non-classical logics and is to be ignored for (classical) nanoCoP.

The predicate `prove(Mat,PathLim,Set,Proof)` implements the start rule (lines 1–7) and iterative deepening (lines 8–12). `Mat` is the (prefixed) matrix generated in the preprocessing step, `PathLim` is the size limit for `Path`, `Set` is a strategy (see Section 2.3), and `Proof` contains the returned (non-clausal) connection proof. The predicate `positiveC(Cla,Cla1)` returns the positive clause `Cla1` of `Cla` (implemented in 7 additional lines of code). The predicate `prove(Cla,Mat,Path,PathI,PathLim,Lem,PreS,VarS,Set,Proof)` implements the axiom (line 13), the decomposition rule (lines 14–19), the reduction rule (lines 20–23, 27–29, 40–41), and the extension rule (lines 20–23, 31–41) of the calculus in Fig. 7. The substitution σ is stored implicitly by Prolog. The predicate `prove_ec(ClaB,Cla1,Mat,ClaB1,Mat1)` calculates extension clauses (lines 42–52). Additional optimization techniques (see Section 2.3) are regularity (line 22), lemmata (lines 24–25) and restricted backtracking (line 39).

5.3. nanoCoP-i and nanoCoP-M

nanoCoP-i and nanoCoP-M are compact Prolog implementations of the non-clausal connection calculus for first-order *intuitionistic* logic (with equality) and for the first-order *modal* logics D, T, S4 and S5 with constant, cumulative and varying domains, respectively [37, 35].

```
        % start rule
(1)     prove(Mat,PathLim,Set,[(I^0)^V:Proof]) :-
(2)         ( member(scut,Set) -> ( append([(I^0)^V^VS:Cla1|_],[!|_],Mat) ;
(3)             member((I^0)^V^VS:Cla,Mat), positiveC(Cla,Cla1) ) -> true ;
(4)             ( append(MatC,[!|_],Mat) -> member((I^0)^V^VS:Cla1,MatC) ;
(5)             member((I^0)^V^VS:Cla,Mat), positiveC(Cla,Cla1) ) ),
(6)         prove(Cla1,Mat,[],[I^0],PathLim,[],PreS,VarS,Set,Proof),
(7)         append(VarS,VS,VarS1), domain_cond(VarS1), prefix_unify(PreS).

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

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

        % decomposition rule
(14)    prove([J^K:Mat1|Cla],MI,Path,PI,PathLim,Lem,PreS,VarS,Set,Proof) :-
(15)        !, member(I^V^FV:Cla1,Mat1),
(16)        prove(Cla1,MI,Path,[I,J^K|PI],PathLim,Lem,PreS1,VarS1,Set,Proof1),
(17)        prove(Cla,MI,Path,PI,PathLim,Lem,PreS2,VarS2,Set,Proof2),
(18)        append(PreS2,PreS1,PreS), append(FV,VarS1,VarS3),
(19)        append(VarS2,VarS3,VarS),  Proof=[J^K:I^V:Proof1|Proof2].

        % reduction and extension rules
(20)    prove([Lit:Pre|Cla],MI,Path,PI,PathLim,Lem,PreS,VarS,Set,Proof) :-
(21)        Proof=[Lit:Pre,I^V:[NegLit:PreN|Proof1]|Proof2],
(22)        \+ (member(LitC,[Lit:Pre|Cla]), member(LitP,Path), LitC==LitP),
(23)        (-NegLit=Lit;-Lit=NegLit) ->
(24)           ( member(LitL,Lem), Lit:Pre==LitL, Proof1=[], I=1, V=[],
(25)             PreS3=[], VarS3=[]
(26)             ;
(27)             member(NegL:PreN,Path), unify_with_occurs_check(NegL,NegLit),
(28)             Proof1=[], I=r, V=[],
(29)             \+ \+ prefix_unify([Pre=PreN]), PreS3=[Pre=PreN], VarS3=[]
(30)             ;
(31)             lit(NegLit:PreN,ClaB,Cla1,Grnd1),
(32)             ( Grnd1=g -> true ; length(Path,K), K<PathLim -> true ;
(33)               \+ pathlim -> assert(pathlim), fail ),
(34)             \+ \+ prefix_unify([Pre=PreN]),
(35)             prove_ec(ClaB,Cla1,MI,PI,I^V^FV:ClaB1,MI1),
(36)             prove(ClaB1,MI1,[Lit:Pre|Path],[I|PI],PathLim,Lem,PreS1,VarS1,
(37)                   Set,Proof1), PreS3=[Pre=PreN|PreS1], append(VarS1,FV,VarS3)
(38)           ),
(39)           ( member(cut,Set) -> ! ; true ),
(40)           prove(Cla,MI,Path,PI,PathLim,[Lit:Pre|Lem],PreS2,VarS2,Set,Proof2),
(41)           append(PreS3,PreS2,PreS), append(VarS2,VarS3,VarS).

        % extension clause (e-clause)
(42)    prove_ec((I^K)^V:ClaB,IV:Cla,MI,PI,ClaB1,MI1) :-
(43)        append(MIA,[(I^K1)^V1:Cla1|MIB],MI), length(PI,K),
(44)        ( ClaB=[J^K:[ClaB2]|_], member(J^K1,PI),
(45)          unify_with_occurs_check(V,V1), Cla=[_:[Cla2|_]|_],
(46)          append(ClaD,[J^K1:MI2|ClaE],Cla1),
(47)          prove_ec(ClaB2,Cla2,MI2,PI,ClaB1,MI3),
(48)          append(ClaD,[J^K1:MI3|ClaE],Cla3),
(49)          append(MIA,[(I^K1)^V1:Cla3|MIB],MI1)
(50)          ;
(51)          (\+member(I^K1,PI);V\==V1) ->
(52)          ClaB1=(I^K)^V:ClaB, append(MIA,[IV:Cla|MIB],MI1) ).
```

Figure 8: The source code of the nanoCoP 2.0, nanoCoP-i 2.0 and nanoCoP-M 2.0 core provers for classical, intuitionistic and modal logic.

The source code of nanoCoP-i 2.0 and nanoCoP-M 2.0 is shown in Fig. 8. The underlined text is added to the nanoCoP code for classical logic. First, nanoCoP-i and nanoCoP-M perform a classical proof search, in which the prefixes of each connection are stored in `PreS`. If the search succeeds, the domain condition is checked using the predicate `domain_cond` (line 7) and the prefixes in `PreS` are unified using the predicate `prefix_unify` (line 7). These predicates are implemented by 18 and between 7 to 22 lines of code (depending on the logic), respectively.

nanoCoP-i 2.0 and nanoCoP-M 2.0 are some of the fastest provers for first-order intuitionistic logic and first-order modal logic (D, T, S4 and S5), respectively [35].

6. Other CoPs — Re-Implementations and Machine Learning

Several other provers are based on or were inspired by the leanCoP prover. More detailed information about these provers can be found in the cited references.

6.1. randoCoP, leanCoP-SiNE, leanCoP-Ω and leanCoP on iPod

randoCoP [40] is a theorem prover for classical first-order logic, which integrates randomized search techniques into the connection prover leanCoP 2.0. By randomly reordering the axioms of the input problem/formula and the literals within its clausal form, the performance of the incomplete search strategies of leanCoP 2.0, such as restricted backtracking, can be improved significantly. A reordering strategy was later also integrated into leanCoP 2.1. At CASC in 2008, randoCoP 1.1 was among the top three provers that return a proof in the core FOF division.

leanCoP-SiNE is a theorem prover for classical first-order logic, which integrates the SiNE preprocessor into leanCoP 2.1. SiNE [41] is an axiom selection system that uses a syntactic approach based on predicate and function symbols in the input problem/formula to select axioms that are (likely) relevant to find a proof. It significantly improves performance on very large problems. At CASC in 2009, leanCoP-SiNE 2.1 ends up in third place in the proof class of the SUMO reasoning prize.

leanCoP-Ω is a prover for classical first-order logic with equality and interpreted functions and predicates for linear integer arithmetic. It combines leanCoP 2.1 with the Omega 1.2 test system [42]. The prover was developed in cooperation with Holger Trölenberg and Thomas Raths. At CASC in 2010, leanCoP-Ω 0.1 wins the first (linear integer arithmetic) TFA division.

leanCoP on iPod [43] is an implementation of leanCoP 2.1 on a (1st-generation) iPod nano. An iPod nano is a very compact portable media/MP3 player with an 80 MHz ARM 7TDMI 32-bit CPU and 32 MB of RAM, which was released in 2005. It shows how low leanCoP's resource requirements, in particular with respect to RAM are.

6.2. Re-Implementations of leanCoP

Re-implementations and extensions of leanCoP include lolliCoP (implemented in the linear logic programming language Lolli) [44], fCoP (implemented in OCaml) [45], "C-leanCoP" (implemented in C) [46], RACCOON/leanCoR (for the description logic ALC) [47], fleanCoP/fnanoCoP (implemented in OCaml) [48], SATCoP (integrating a SAT solver) [49], meanCoP (implemented in Rust) [50], Connect++ (in C++) [51], and pyCoP/ipyCoP/mpyCoP (in Python) [52].

6.3. Machine Learning CoPs

At TABLEAUX 2011, the MaLeCoP [53] prover was presented, one of the first implementations that integrates *Machine Learning (ML)* into a theorem prover. It was the starting point for a whole series of ML provers based on or inspired by leanCoP. Among them are MaLeCoP (extends leanCoP by integrating ML techniques) [53] FEMaLeCoP (an optimized version of MaLeCoP) [54], rlCoP (reinforcement learning using Monte Carlo search) [55], plCoP (extending rlCoP and implemented in Prolog) [56], graphCoP (uses graph neural network models) [57], monteCoP (using Monte Carlo Tree search) [48], lazyCoP (implements lazy paramodulation and uses deep neural networks) [58], and FLoP (geared towards finding longer proofs) [59].

7. Conclusion

What started 20 years ago with seven lines of Prolog code, has established itself as a source and starting point for the development of automated theorem provers for classical and non-classical logics, as well as for the integration of machine learning into theorem provers. This happened despite the fact that at the beginning, reception towards leanCoP was controversial and the author faced some resistance even within his own community. Meanwhile, leanCoP has shown that compact Prolog implementations are not only more flexible, extendable, "correct" and easier to maintain than systems with ten thousands lines of low level language code, but they can have state of the art performance. nanoCoP is perhaps the fastest prover that works on the original formula structure, combining the advantages of more natural sequent and tableau provers with the systematic and goal-oriented proof search of connection provers.[10]

Perhaps the philosophy behind leanCoP is best summarized by Tony Hoare [60]: *"I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws which underlie the complex phenomena of nature."*

Acknowledgments

The author would like to thank everyone who contributed directly or indirectly to the development of the provers presented in this article, in particular Wolfgang Bibel for his continued, generous support and (in alphabetical order) Christoph Benzmüller, Michael Färber, Christoph Kreitz, Thomas Raths, Geoff Sutcliffe, Holger Trölenberg, Josef Urban, Arild Waaler and Lincoln Wallen. Thanks also go to the anonymous reviewers for their helpful comments.

References

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

[10] The source code of leanCoP and all other provers presented in this paper can be downloaded at www.leancop.de.

[2] W. Bibel, Matings in matrices, Communications of the ACM 26 (1983) 844–852.
[3] W. Bibel, Automated Theorem Proving, Artificial intelligence, F. Vieweg und Sohn, 1987.
[4] B. Beckert, J. Posegga, leanTAP: Lean tableau-based deduction, Journal of Automated Reasoning 15 (1995) 339–358.
[5] R. Letz, G. Stenz, Model elimination and connection tableau procedures, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Elsevier Science, Amsterdam, 2001, pp. 2015–2112.
[6] G. Gentzen, Untersuchungen über das Logische Schließen, Mathematische Zeitschrift 39 (1935) 176–210, 405–431.
[7] R. M. Smullyan, First-Order Logic, Ergebnisse der Mathematik und ihrer Grenzgebiete, Springer-Verlag, Berlin, Heidelberg, New York, 1971.
[8] R. Hähnle, Tableaux and related methods, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Elsevier Science, Amsterdam, 2001, pp. 101–178.
[9] J. A. Robinson, A machine-oriented logic based on the resolution principle, Journal of ACM 12 (1965) 23–41.
[10] W. McCune, L. Wos, Otter – the CADE-13 competition incarnations, Journal of Automated Reasoning 18 (1997) 211–220.
[11] J. Urban, MPTP 0.2: Design, implementation, and initial experiments, Journal of Automated Reasoning 37 (2006) 21–43.
[12] J. Otten, Restricting backtracking in connection calculi, AI Commun. 23 (2010) 159–182.
[13] J. Otten, leanCoP 2.0 and ileanCoP 1.2: High performance lean theorem proving in classical and intuitionistic logic, in: A. Armando, P. Baumgartner, G. Dowek (Eds.), IJCAR 2008, volume 5195 of *LNAI*, Springer, 2008, pp. 283–291.
[14] M. E. Stickel, A Prolog technology theorem prover: Implementation by an extended Prolog compiler, Journal of Automated Reasoning 4 (1988) 353–380.
[15] G. Sutcliffe, The CADE-21 automated theorem proving system competition, AI Commun. 21 (2008) 71–81.
[16] L. Kovacs, A. Voronkov, First-Order Theorem Proving and Vampire, in: N. Sharygina, H. Veith (Eds.), Proceedings of the 25th International Conference on Computer Aided Verification, number 8044 in LNAI, Springer, 2013, pp. 1–35.
[17] D. van Dalen, Intuitionistic Logic, John Wiley & Sons, 2017, pp. 224–257.
[18] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions, EATCS Series, Springer, Heidelberg, 2004.
[19] R. L. Constable, et al., Implementing Mathematics with the NuPRL proof development system, Prentice–Hall, Englewood Cliffs, NJ, 1986.
[20] L. A. Wallen, Automated Deduction in Nonclassical Logics, MIT Press, Cambridge, 1990.
[21] J. Otten, Clausal connection-based theorem proving in intuitionistic first-order logic, in: B. Beckert (Ed.), TABLEAUX 2005, volume 3702 of *LNAI*, Springer, 2005, pp. 245–261.
[22] A. Waaler, Connections in nonclassical logics, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Elsevier Science, Amsterdam, 2001, pp. 1487–1578.
[23] R. Statman, Intuitionistic propositional logic is polynomial-space complete, Theoretical Computer Science 9 (1979) 67–72.
[24] S. A. Cook, The complexity of theorem-proving procedures, in: Third Annual ACM Symposium on Theory of Computing, ACM, New York, 1971, pp. 151–158.

[25] P. Blackburn, J. van Bentham, F. Wolter, Handbook of Modal Logic, Elsevier, Amsterdam, 2006.
[26] M. Fitting, R. L. Mendelsohn, First-Order Modal Logic, Kluwer, Dordrecht, 1998.
[27] M. Fitting, Proof Methods for Modal and Intuitionistic Logics, D. Reidel, Dordrecht, 1983.
[28] J. Otten, Implementing connection calculi for first-order modal logics, in: K. Korovin, S. Schulz, E. Ternovska (Eds.), IWIL 2012, volume 22 of *EPiC*, EasyChair, 2012, pp. 18–32.
[29] J. Otten, MleanCoP: A connection prover for first-order modal logic, in: S. Demri, D. Kapur, C. Weidenbach (Eds.), IJCAR 2014, volume 8562 of *LNAI*, Springer, 2014, pp. 269–276.
[30] K. Gödel, An interpretation of the intuitionistic sentential logic, in: J. Hintikka (Ed.), The Philosophy of Mathematics, Oxford University Press, Oxford, 1969, pp. 128–129.
[31] J. Otten, nanoCoP: A non-clausal connection prover, in: N. Olivetti, A. Tiwari (Eds.), IJCAR 2016, volume 9706 of *LNAI*, Springer, 2016, pp. 302–312.
[32] J. Otten, W. Bibel, Advances in connection-based automated theorem proving, in: M. Hinchey, J. P. Bowen, E.-R. Olderog (Eds.), Provably Correct Systems, NASA Monographs in Systems and Software Engineering, Springer, Cham, 2017, pp. 211–241.
[33] T. Raths, J. Otten, The QMLTP problem library for first-order modal logics, in: B. Gramlich, et al. (Eds.), IJCAR 2012, volume 7364 of *LNAI*, Springer, 2012, pp. 454–461.
[34] C. Benzmüller, J. Otten, T. Raths, Implementing and evaluating provers for first-order modal logics, in: L. De Raedt, et al. (Eds.), 20th European Conference on Artificial Intelligence, ECAI 2012, IOS Press, Amsterdam, 2012, pp. 163–168.
[35] J. Otten, The nanoCoP 2.0 connection provers for classical, intuitionistic and modal logics, in: A. Das, S. Negri (Eds.), TABLEAUX 2021, volume 12842 of *LNAI*, Springer, 2021, pp. 236–249.
[36] J. Otten, nanoCoP: Natural non-clausal theorem proving, in: C. Sierra (Ed.), Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, Sister Conference Best Paper Track, IJCAI, 2017, pp. 4924–4928.
[37] J. Otten, Non-clausal connection calculi for non-classical logics, in: R. Schmidt, C. Nalon (Eds.), TABLEAUX 2017, volume 10501 of *LNAI*, Springer, 2017, pp. 209–227.
[38] J. Otten, A non-clausal connection calculus, in: K. Brünnler, G. Metcalfe (Eds.), TABLEAUX 2011, volume 6793 of *LNAI*, Springer, 2011, pp. 226–241.
[39] W. Bibel, J. Otten, From Schütte's formal systems to modern automated deduction, in: R. Kahle, M. Rathjen (Eds.), The Legacy of Kurt Schütte, Springer, Cham, 2020, pp. 217–251.
[40] T. Raths, J. Otten, randoCoP: randomizing the proof search order in the connection calculus, in: B. Konev, R. Schmidt, S. Schulz (Eds.), PAAR 2008, volume 373 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2008, pp. 94–102.
[41] K. Hoder, A. Voronkov, Sine qua non for large theory reasoning, in: N. Bjørner, V. Sofronie-Stokkermans (Eds.), CADE-23, volume 6803 of *LNCS*, Springer, 2011, pp. 299–314.
[42] W. Pugh, The omega test: A fast and practical integer programming algorithm for dependence analysis, Communications of the ACM 31 (1992) 4–13.
[43] J. Otten, The pocket reasoner – automatic reasoning on small devices., in: E. B. Johnsen, I. C. Yu (Eds.), Norwegian Informatics Conference, NIK 2018, Open Journal Systems, 2018.
[44] J. S. Hodas, N. Tamura, lolliCoP - A linear logic implementation of a lean connection-method theorem prover for first-order classical logic, in: R. Goré, A. Leitsch, T. Nipkow (Eds.), IJCAR 2001, volume 2083 of *LNCS*, Springer, 2001, pp. 670–684.

[45] C. Kaliszyk, J. Urban, J. Vyskočil, Certified Connection Tableaux Proofs for HOL Light and TPTP, in: Proceedings of the 2015 Conference on Certified Programs and Proofs (CPP 2015), ACM, 2015, pp. 59–66.
[46] C. Kaliszyk, Efficient low-level connection tableaux, in: H. de Nivelle (Ed.), TABLEAUX 2015, volume 9323 of *LNCS*, Springer, 2015, pp. 102–111.
[47] D. M. Filho, F. Freitas, J. Otten, RACCOON: A connection reasoner for the description logic ALC, in: T. Eiter, D. Sands (Eds.), LPAR-21, volume 46 of *EPiC*, EasyChair, 2017, pp. 200–211.
[48] M. Färber, C. Kaliszyk, J. Urban, Machine learning guidance for connection tableaux, Journal of Automated Reasoning 65 (2021) 287–320.
[49] M. Rawson, G. Reger, Eliminating models during model elimination, in: A. Das, S. Negri (Eds.), TABLEAUX 2021, volume 12842 of *LNCS*, Springer, 2021, pp. 250–265.
[50] M. Färber, Connection Provers in Rust, 2022. URL: https://github.com/01mf02/cop-rs.
[51] S. B. Holden, Connect++: A new automated theorem prover based on the connection calculus, in: J. Otten, W. Bibel (Eds.), Automated Reasoning with Connection Calculi, AReCCa 2023, CEUR Workshop Proceedings, CEUR-WS.org, to appear, pp. 95–106.
[52] F. Rømming, J. Otten, S. B. Holden, Connections: Markov decision processes for classical, intuitionistic, and modal connection calculi, in: J. Otten, W. Bibel (Eds.), Automated Reasoning with Connection Calculi, AReCCa 2023, CEUR Workshop Proceedings, CEUR-WS.org, to appear, pp. 107–118.
[53] J. Urban, J. Vyskocil, P. Stepánek, MaLeCoP machine learning connection prover, in: K. Brünnler, G. Metcalfe (Eds.), TABLEAUX 2011, volume 6793 of *LNCS*, Springer, 2011, pp. 263–277.
[54] C. Kaliszyk, J. Urban, FEMaLeCoP: Fairly efficient machine learning connection prover, in: M. Davis, A. Fehnker, A. McIver, A. Voronkov (Eds.), LPAR-20, volume 9450 of *LNAI*, Springer, 2015, pp. 88–96.
[55] C. Kaliszyk, J. Urban, H. Michalewski, M. Olšák, Reinforcement Learning of Theorem Proving, in: Advances in Neural Information Processing Systems, volume 31, Curran Associates, Inc., 2018.
[56] Z. Zombori, J. Urban, C. E. Brown, Prolog technology reinforcement learning prover, in: N. Peltier, V. Sofronie-Stokkermans (Eds.), IJCAR 2020, volume 12167 of *LNAI*, Springer, 2020, pp. 489–507.
[57] M. Olšák, C. Kaliszyk, J. Urban, Property invariant embedding for automated reasoning, in: G. D. Giacomo, et al. (Eds.), ECAI 2020, volume 325 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, Amsterdam, 2020, pp. 1395–1402.
[58] M. Rawson, G. Reger, lazyCoP: Lazy Paramodulation Meets Neurally Guided Search, in: A. Das, S. Negri (Eds.), TABLEAUX 2021, volume 12842 of *LNAI*, Springer, 2021, pp. 187–199.
[59] Z. Zombori, A. Csiszárik, H. Michalewski, C. Kaliszyk, J. Urban, Towards finding longer proofs, in: A. Das, S. Negri (Eds.), TABLEAUX 2021, volume 12842 of *LNCS*, Springer, 2021, pp. 167–186.
[60] C. A. R. Hoare, The emperor's old clothes, Communications of the ACM 24 (1981) 75–83.