Article Submitted to Journal of Symbolic Computation

leanCoP: Lean Connection-Based Theorem Proving

Jens Otten and Wolfgang Bibel

Fachgebiet Intellektik, Fachbereich Informatik, Darmstadt University of Technology, Alexanderstr. 10, 64283 Darmstadt, Germany

Abstract

The Prolog program

```
"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)."
```

implements a theorem prover for classical first-order (clausal) logic which is based on the connection calculus. It is sound and complete (provided that an arbitrarily large I is iteratively given), and demonstrates a comparatively strong performance.

1. Introduction

The connection calculus (Bibel, 1983, 1987, 1993), the connection tableau calculus (Letz *et al.*, 1994) and the similar model elimination calculus (Loveland, 1968) are popular and successful proof procedures because of their goal-oriented search strategy. Several proof systems based on one of these approaches have been developed, e.g. KoMeT (Bibel *et al.*, 1994), Setheo (Letz *et al.*, 1992; Moser *et al.*, 1997) and Meteor (Astrachan and Loveland, 1991) to name a few. All these systems have shown to be an appropriate basis to automate formal reasoning in classical first-order logic.

The Prolog program shown in the abstract has been developed in the context of a graduate course about "Automated Deduction". Its main purpose was to demonstrate a small and easy to use implementation of the (clausal) connection calculus which can easily be understood and modified by the students themselves. It turned out that the implementation is not only very compact but also shows a surprisingly good performance.

The interest in *lean theorem proving* arised after the theorem prover leanTAP (Beckert and Posegga, 1995) became popular. leanTAP implements a free-variable semantic tableau calculus and its minimal version consists only of eight lines of Prolog code. leanTAP showed that it is possible to reach considerable performance by using very compact code, thus making lean theorem provers an interesting alternative for applications where state-of-the-art performance is not required. In contrast to huge proof systems using a lot of sophisticated techniques, lean theorem provers can easily be modified and adapted for special purposes. Furthermore it is much easier to verify a few lines of Prolog code than to verify thousands of lines of e.g. C code.

leanCoP consists only of three Prolog clauses. Like leanTAP the minimal version is only a few lines long. The underlying calculus though is entirely different: the connectedness condition needs a different kind of implementation techniques. Whereas leanTAP performs well on formulas in negation normal form, the performance can be considerably improved by the connection based approach of leanCoP in particular for formulas in clausal form. leanCoP even finds proofs for a number of problems which cannot be solved by current state-of-the-art theorem provers. Thus our prover follows the tradition of lean theorem proving, giving lean yet efficient code.

Outline of the Paper

In Section 2 we will explain in detail the Prolog source code of leanCoP as well as some basic techniques used within the code. Section 3 presents performance results obtained by extensive experimental tests on problems in the TPTP library. We compare leanCoP with three other well-known theorem provers based on different calculi: the lean semantic tableau prover leanTAP (Beckert and Posegga, 1995), the Prolog technology theorem prover PTTP (Stickel, 1992), and the resolution based theorem prover OTTER (McCune, 1994). In Section 4 we will describe an easy way to refine the depth-bounded search of leanCoP. In Section 5 we will prove completeness and correctness of leanCoP. To this end we transform the underlying connection calculus stepwisely into a purely declarative Prolog program. We conclude with a short summary, some remarks on related work, and a brief outlook to further research in Section 6.

We assume the reader to be familiar with the basic ideas of Prolog and the connection calculus. See Clocksin and Mellish (1981) for an introduction to Prolog and Bibel (1993) for an introduction to the connection calculus.

2. The Program

Our prover is based on the simplest version of a *connection calculus* (Bibel, 1983, 1987, 1993) and realizes a proof procedure for (full) first-order clause logic. In contrast to (connection) tableau calculi which generate a number of intermediate formulas from the original one, calculi based on the *connection method* operate

exclusively on a single copy of the given formula. If one abstracts from this difference which, however, is important for efficiency, a connection calculus can be considered as constructing a *connection tableau* (Letz *et al.*, 1994) where open subgoals are selected in a depth-first way.

The process starts by selecting a *start clause* before *extension steps* and *reduction steps* are repeatedly applied. Whereas the extension step connects a subgoal literal to a complementary literal of a new clause instance, the reduction step connects it to a complementary literal of the so-called *active path*. The extension step actually realizes the goal-oriented proof search.

To prove a formula we first need to translate the given first-order formula into a set of clauses. We use the positive representation throughout the paper, i.e. we *prove* a formula in disjunctive normal form which is equivalent to refuting its negation in conjunctive normal form. Consider for example the following formula, which is problem 21 of Pelletier (1985): $(\exists X(p \Rightarrow f(X)) \wedge \exists X(f(X) \Rightarrow p)) \Rightarrow \exists X(p \Leftrightarrow f(X))$. The translation to disjunctive skolemized normal form yields $(p \wedge \neg f(a)) \vee (f(b) \wedge \neg p) \vee (p \wedge f(X)) \vee (\neg p \wedge \neg f(X))$ which can be directly represented by the clause set or *matrix* $\{\{p, \neg f(a)\}, \{f(b), \neg p\}, \{p, f(X)\}, \{\neg p, \neg f(X)\}\}$.

We will use Prolog lists to represent sets, Prolog terms to represent atomic formulas, Prolog variables to represent first-order variables, and "`-`" to represent the negation "¬". Thus the above clause set is represented by the Prolog list `[[p,-f(a)],[f(b),-p],[p,f(X)],[-p,-f(X)]]` called `Mat` in the following program.

We use the Prolog predicates `prove/2` (with two arguments) and `prove/4` (with four arguments) to implement leanCoP. The first Prolog clause of `prove/2` selects a start clause from the given clause set. The two Prolog clauses of `prove/4` realize the extension and reduction steps. A second Prolog clause of `prove/2` (discussed in Section 2.3) can be added to realize an iterative deepening proof search which is necessary to gain completeness.

2.1. Selecting a Start Clause

The Prolog predicate

```
prove(Mat,PathLim)
```

succeeds if there is a connection proof for the clause set `Mat` whose active path lengths for all extension steps to first-order clauses, i.e. clauses which contain at least one variable, are limited by `PathLim`.

```
prove(Mat,PathLim) :-
     append(MatA,[Cla|MatB],Mat), \+member(-_,Cla),
     append(MatA,MatB,Mat1),
     prove([!],[[-!|Cla]|Mat1],[],PathLim).
```

A start clause `Cla` is selected by the (built-in) predicate `append`. Usually `append` is used to append two lists, e.g. `append([a],[b,c],L)` yields `L=[a,b,c]`. If the first two arguments are uninstantiated, all possible solutions for them are given on backtracking. For example `append(A,[X|B],[a,b,c]),append(A,B,C)` will produce the three solutions `X=a,C=[b,c]`, `X=b,C=[a,c]`, and `X=c,C=[a,b]`. It realizes an easy way to successively select an element from a list, returning the list without this element.

Since it is sufficient to consider only positive start clauses, we will only select clauses `Cla` which do not contain any negative literals, i.e. no literals of the form $\neg q$. Only if `Cla` is a positive clause the goal `\+member(-_,Cla)` succeeds.

Afterwards the predicate `prove/4` which realizes extension and reduction step is called. The actual proof search is started using the special literal "!" as the root. Instead of taking the selected start clause as the first subgoal clause, we start the proof search with the subgoal containing only "!" and add the literal "¬!" to the original start clause. The literal "!" should not occur in the clause set `Mat`, so that the original start clause will be used for the first extension step. This kind of initial step is necessary in order to allow copies of the start clause later on.* Note that the "!" as used in this context is *not* a Prolog cut.

To prove our previous example using a maximal path length of 2, we have to call the goal `prove([[p,-f(a)],[f(b),-p],[p,f(X)],[-p,-f(X)]],2)` which will succeed. Hence our original formula is valid.

2.2. The Extension and the Reduction Step

The Prolog predicate

```
prove(Cla,Mat,Path,PathLim)
```

succeeds if there is a proof for the clause of open subgoals `Cla` using the clauses in `Mat` and the active `Path` where the active path lengths for all extension steps to first-order clauses are limited by `PathLim`.

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

If the clause of open subgoals is empty, we do not have to perform any further search. In this case the first clause of `prove/4` will succeed. Otherwise the second clause of the predicate `prove/4` matches.

```
prove([Lit|Cla],Mat,Path,PathLim) :-
     (-NegLit=Lit;-Lit=NegLit) ->
        ( member(NegLit,Path);
          append(MatA,[Cla1|MatB],Mat), copy_term(Cla1,Cla2),
          append(ClaA,[NegLit|ClaB],Cla2),append(ClaA,ClaB,Cla3),
```

*Starting with `prove(Cla,Mat1,[],PathLim)` instead results in incompleteness.

```
    ( Cla1==Cla2 -> append(MatB,MatA,Mat1);
                    length(Path,K), K<PathLim,
                    append(MatB,[Cla1|MatA],Mat1)
    ), prove(Cla3,Mat1,[Lit|Path],PathLim)
  ), prove(Cla,Mat,Path,PathLim).
```

Now we try to find a solution for the literal `Lit` from the open subgoals. After `NegLit` is bound to the negation of `Lit`, it is checked whether an application of a reduction step is possible, i.e. whether `NegLit` unifies with an element of `Path`, using the (built-in) predicate `member`. For this sound unification has to be used.† If a reduction step is performed we skip to the last line where `prove/4` is called to find solutions for the remaining subgoals in `Cla`. Otherwise an extension step is performed which will first select a clause `Cla1` from `Mat` using `append` as explained before. A copy `Cla2` of the clause `Cla1` is made (where all variables in `Cla2` are renamed)‡ using the (built-in) predicate `copy_term`. And an element of `Cla2` which unifies with `NegLit` is selected using again our "append technique". Again sound unification has to be used for unifying `NegLit` with an element of `Cla2`. `Cla3` is bound to the remaining literals in `Cla2`.

If the clauses `Cla1` and `Cla2` are (syntactically) identical, i.e. do not contain any variables, `Mat1` is bound to the remaining clauses in `Mat` (without the clause `Cla1`). Otherwise the clause `Cla1` is included in the set `Mat1`, after it has been checked that the length `K` of the active `Path` does not exceed the limit `PathLim`. Limiting the active path is necessary to achieve completeness within Prolog's incomplete depth-first search strategy. "`Goal1 -> Goal2 ; Goal3`" implements the if-then-else construct in Prolog. It succeeds if either `Goal1` succeeds and then `Goal2` succeeds or else if `Goal1` fails, and then `Goal3` succeeds. There is no backtracking over `Goal1` once it has succeeded (i.e. there is an *implicit cut*). Note that we slightly reordered the clauses in `Mat1`. Clauses in `MatB`, which have not been investigated during the current extension step, are placed ahead of all other clauses in `Mat1`. In general this leads to a better arrangement of the search space.

Finally `prove/4` is called to find solutions for the new clause of open subgoals `Cla3`, where `Lit` has been added to `Path`, and for the remaining open subgoals in `Cla`.

2.3. Iterative Deepening

If the following clause is added after the first clause of `prove/2`

```
prove(Mat,PathLim) :-
     nonground(Mat), PathLim1 is PathLim+1, prove(Mat,PathLim1).
```

†In eclipse Prolog sound unification is switched on with `set_flag(occur_check,on)`.

‡Hence it is not necessary for the set of input clauses to have disjoint variables.

iterative deepening on the proof search depth, i.e. the length of the active path, is performed yielding completeness for first-order logic. The (built-in) predicate `nonground(Mat)` succeeds if `Mat` does contain at least one (first-order) variable.[§] In this case `Mat` represents a first-order formula and the limit `PathLim` is increased before the proof search is restarted. Otherwise, if `Mat` represents a variable-free or *ground* formula, the predicate fails and the clause set `Mat` is not valid. Remember that we do not check the length of `Path` for variable-free clauses, so we do not need to increase `PathLim` for variable-free formulas. This immediately yields a decision procedure for propositional logic.

For our previous example we start the proof search using iterative deepening by `prove([[p,-f(a)],[f(b),-p],[p,f(X)],[-p,-f(X)]],1)`. There is no proof with a path limit of 1, but the second proof attempt using a path limit of 2 will eventually succeed.

3. Performance

We have tested leanCoP on the problems contained in the current version 2.3.0 of the TPTP library (Sutcliffe and Suttner, 1998). We have tested it on all 2193 propositional and first-order problems in clausal form which are known to be valid (or unsatisfiable using negative representation) and all 7 propositional problems known to be invalid (or satisfiable). No reordering of clauses or literals has been done. When transforming the formulas into an appropriate input format for leanCoP we translated literals of the form `++q` into `-q` and literals of the form `--q` into `q`, respectively, since we use a positive representation.

All tests were performed on a SUN Ultra10 with 128 Mbytes memory using eclipse Prolog version 3.5.2. When compiling leanCoP the generation of debug information has been switched off using "`nodbgcomp`". The time limit for all proof attempts was 300 seconds.

Table 1: Overall performance of leanCoP on the TPTP library

number of all tested problems	problems solved within 300 seconds			
2200 (100%)	750 (34.1%)			
	390	185	121	54
	less than 1s	1s to 10s	10s to 100s	100s to 300s

Even though a lot of the problems are rather hard, leanCoP was able to solve 750 problems, 390 of them in less than one second (see Table 1). In the TPTP library the difficulty of each problem is rated from 0.0 to 1.0 relative to state-of-the-art theorem provers. A rating of 0.0 means that all state-of-the-art provers can solve the problem, a rating of 1.0 means that no state-of-the-art prover can

[§]In some Prolog dialects `\+ground(Mat)` has to be used for this purpose instead.

solve it. leanCoP solves more than half of the problems rated 0.0. Table 2 shows the number of solved problems classified with respect to the problem rating. Problems rated "?" are those problems which are not rated yet.

Table 2: Performance on TPTP library classified with respect to problem rating

rating	0.0	0.01 to 0.32	0.33 to 0.65	0.66 to 0.99	1.0	?
total	1308	189	326	165	53	159
solved	673 (51%)	26 (14%)	29 (9%)	5 (3%)	0 (0%)	17 (11%)

leanCoP is able to solve 60 problems rated higher than 0.0. They are compiled in Table 3. For each of these problems its name and rating is given as well as the timings in seconds. Five of the problems in Table 3 are rated 0.67 which means that most state-of-the-art provers cannot prove them. Four of them are within the field theory domain (FLD), the other (PUZ0034-1.004) is the problem to place 4 queens on a 4×4 chess board, so that no queen can attack another one.

3.1. leanCoP Compared to Otter, Pttp, and lean*TAP*

We have compared leanCoP with three other well-known theorem provers: Otter 3.1 (and Mace 1.4), Pttp (Prolog version 2e), and lean*TAP* (version 2.3). Otter (McCune, 1994) is a theorem prover based on resolution and paramodulation which has been very successful in proving difficult mathematical problems. Pttp (Stickel, 1988, 1992) is an implementation of the model elimination theorem-proving procedure that extends Prolog to the full first-order calculus. It achieves a high inference rate by compiling the input formula into a Prolog program. Sound unification, iterative deepening, and the reduction rule are added to gain a complete search procedure. It uses an inference-bounded proof search (see also Section 4). lean*TAP* (Beckert and Posegga, 1995; Posegga and Schmitt, 1999) is a first-order theorem prover based on free-variable semantic tableaux. Its very compact Prolog implementation achieves a surprisingly good performance, in particular for input formulas in non-clausal form.

The timings of Otter on the TPTP library are regularly published (Argonne National Laboratory, 2000). They were obtained on a 400 MHz Linux machine which should be slightly faster than our machine. The timings of Pttp and lean*TAP* were obtained on our SUN Ultra10 using eclipse Prolog. All problems have been converted into Pttp and lean*TAP* syntax by using the tools provided with the TPTP library. Again no reordering of clauses or literals has been done. The overall performance of these three provers and leanCoP is shown in Table 4.

As expected Otter solves the largest number of problems: 1602 out of the tested 2200 problems, most of them within 1 second. 249 of the solved problems are rated difficult, i.e. higher than 0.0. On 59 problems Otter failed because of an empty set-of-support ("sos") or due to a lack of memory. Otter cannot

Table 3: TPTP problems solved by leanCoP and rated greater than 0.0

Problem	Rating	OTTER	PTTP	leanCoP
BOO012-1	(0.17)	3	51.04	28.53
CAT003-2	(0.50)	>300	>300	34.87
CAT003-3	(0.11)	>300	113.48	6.76
CAT012-4	(0.17)	1	0.42	46.21
COL002-3	(0.33)	>300	0.07	0.03
FLD013-1	(0.67)	>300	26.07	1.31
FLD023-1	(0.33)	>300	0.47	1.66
FLD025-1	(0.67)	>300	25.71	1.31
FLD030-1	(0.33)	1	0.10	0.08
FLD030-2	(0.33)	>300	0.13	1.28
FLD037-1	(0.33)	>300	0.91	4.45
FLD060-1	(0.67)	>300	4.53	1.59
FLD061-1	(0.67)	>300	5.78	1.91
FLD067-1	(0.33)	>300	0.16	3.95
FLD070-1	(0.33)	>300	0.15	6.91
FLD071-3	(0.33)	2	0.12	1.03
GEO026-3	(0.11)	2	>300	129.27
GEO041-3	(0.22)	1	5.65	296.70
GRP008-1	(0.22)	1	95.23	2.31
LCL045-1	(0.20)	119	0.41	1.78
LCL097-1	(0.20)	1	0.85	0.75
LCL111-1	(0.20)	1	0.11	0.25
LCL130-1	(0.20)	1	0.27	0.03
LCL195-1	(0.20)	sos-empty	0.57	27.00
NUM283-1.005	(0.20)	1	0.37	0.44
NUM284-1.014	(0.20)	1	>300	290.56
PLA004-1	(0.40)	>300	>300	12.44
PLA004-2	(0.40)	>300	>300	18.69
PLA005-1	(0.40)	>300	>300	1.38
PLA005-2	(0.40)	>300	>300	0.38
PLA007-1	(0.40)	>300	10.82	0.44
PLA009-1	(0.40)	>300	>300	0.19
PLA009-2	(0.40)	>300	>300	6.62
PLA011-1	(0.40)	>300	>300	0.44
PLA011-2	(0.40)	>300	>300	1.38
PLA012-1	(0.40)	>300	>300	211.88
PLA013-1	(0.40)	>300	>300	0.75
PLA014-1	(0.40)	>300	>300	6.56
PLA014-2	(0.40)	>300	>300	6.88
PLA016-1	(0.40)	>300	5.78	0.25
PLA019-1	(0.40)	>300	9.59	0.19
PLA021-1	(0.40)	>300	>300	0.56
PLA022-1	(0.40)	>300	1.35	1.19
PLA022-2	(0.40)	>300	0.14	0.12
PLA023-1	(0.40)	>300	>300	231.69
PUZ034-1.004	(0.67)	sos-empty	2.86	35.81
RNG006-2	(0.20)	5	0.13	0.94
RNG040-1	(0.11)	1	0.16	0.06
RNG040-2	(0.22)	1	1.30	0.75
RNG041-1	(0.22)	1	0.41	159.12
SET016-7	(0.12)	>300	0.81	183.31
SET018-7	(0.12)	>300	0.86	187.06
SET060-6	(0.12)	1	0.58	0.62
SET060-7	(0.12)	1	0.63	0.69
SET152-6	(0.12)	1	2.24	46.50
SET153-6	(0.12)	>300	2.20	9.62
SET187-6	(0.38)	>300	>300	238.06
SET231-6	(0.12)	>300	0.64	170.50
SYN048-1	(0.20)	1	0.01	0.01
SYN311-1	(0.20)	sos-empty	6.51	176.69

Table 4: Overall performance of OTTER, PTTP, leanTAP, and leanCoP

	OTTER	PTTP	leanTAP	leanCoP
solved (total)	1602	999	137	750
0 to <1 second	1209	590	110	390
1 to <10 seconds	142	295	10	185
10 to <100 seconds	209	77	16	121
100 to <200 seconds	31	26	0	31
200 to 300 seconds	11	11	1	23
problems rated 0.0	1230	851	130	673
problems rated >0.0	249	121	2	60
problems rated ?	123	27	5	17
proved	1595	999	135	745
refuted	7	0	2	5
timeout (>300 seconds)	539	1201	1978	1450
failed (sos/memory)	59	0	85	0

solve 39 of the 60 difficult problems solved by leanCoP which are shown in Table 3. Most of these problems are within the field theory domain (FLD) and the planning domain (PLA). PTTP solves 999 out of the tested 2200 problems, 121 of them are rated higher than 0.0. leanTAP only solves 135 problems, two of them (FLD067-1 and SYN048-1) are rated "difficult". leanCoP solves every problem which is solved by leanTAP except problem SYN350-1.

The problems in the TPTP library are categorized in 28 different domains, e.g. algebra (ALG), category theory (CAT), combinatory logic (COL), field theory (FLD), geometry (GEO), group theory (GRP), logic calculi (LCL), planning (PLA), puzzles (PUZ), set theory (SET), syntactic (SYN). See Sutcliffe and Suttner (1998) for a detailed description. Table 5 shows the number of problems each prover has successfully solved within each domain. The last two columns are explained in the next section.

OTTER solves the largest number of problems in most domains. Within the FLD domain PTTP solves more problems than all other provers. leanCoP solves 25 problems in the PLA domain which is considerably more than solved by OTTER (5 problems), PTTP (11 problems), and leanTAP (0 problems). Due to its goal-oriented connection-based approach leanCoP in general performs good on *Horn* problems, i.e. problems containing at most one negated literal in each clause. It performs rather bad on problems containing (only) equality since no special techniques for dealing with equality have been integrated into leanCoP.

3.2. leanCoP on Problems of CASC-17

CASC is a competition where the performance of sound, fully automatic first-order theorem proving systems is evaluated. We have run leanCoP on all 135 valid (original) problems in clausal form selected for the CASC-17. The problems were taken from the TPTP library where the clause order has been changed randomly.

Table 5: Performance on TPTP library ordered with respect to problem domains

Domain	OTTER	PTTP	leanTAP	leanCoP	leanCoP$_i$	leanCoP$_{(i)}$
ALG	4	0	0	0	0	0
ANA	0	0	0	0	0	0
BOO	59	15	0	8	8	11
CAT	45	26	0	21	25	28
CID	2	0	0	0	0	0
CIV	11	6	0	2	0	2
COL	94	53	0	45	49	49
COM	5	5	0	5	5	5
FLD	68	92	2	37	64	64
GEO	86	53	1	25	47	48
GRA	1	1	1	1	0	1
GRP	238	93	2	83	80	86
HEN	60	28	0	8	15	15
KRS	9	8	3	7	5	7
LAT	19	1	0	1	1	1
LCL	272	129	35	99	118	118
LDA	13	1	0	0	1	1
MGT	0	0	0	0	0	0
MSC	9	7	1	7	5	7
NUM	27	21	4	18	20	21
PLA	5	11	0	25	11	25
PRV	7	4	0	4	4	5
PUZ	45	27	12	28	22	28
RNG	55	19	0	16	17	17
ROB	14	4	0	1	4	4
SET	139	111	4	52	60	66
SYN	310	279	71	252	225	256
TOP	5	5	1	5	4	5
proved	1595	999	135	745	790	865
refuted	7	0	2	5	0	5
total	1602	999	137	750	790	870

leanCoP is able to solve 10 out of the 135 tested problems. They are compiled in Table 6. More than half of the solved problems are from the planning domain (PLA). OTTER was able to solve 14 out of the 135 tested problems. PTTP solves 6 problems whereas leanTAP does not solve any selected problem.

The selected problems are divided into classes according to the problem characteristics. The MIX class contains mixed "really-non-propositional theorems" in clausal form. Mixed means Horn and non-Horn problems, with or without equality, but not unit equality problems. Really-non-propositional means problems with an infinite Herbrand universe. leanCoP solves 9 problems which belong to the MIX class (all solved problems except COL020-1). That is one more problem than OTTER was able to solve in this class. For the final results the proof systems have been ranked according to the number of solved problems and the

Table 6: Problems of CASC-17 solved by leanCoP

Problem	Rating	Time (sec)
CAT002-4	(0.17)	8.58
CAT003-2	(0.50)	7.15
COL020-1	(0.00)	0.05
PLA004-2	(0.40)	82.31
PLA005-2	(0.40)	0.36
PLA009-2	(0.40)	1.13
PLA011-2	(0.40)	0.30
PLA014-1	(0.40)	89.34
PLA019-1	(0.40)	4.61
SYN311-1	(0.20)	181.77

average runtime for successful solutions. Table 7 shows an extract from the final result summary for the MIX class where the result of leanCoP has been included. The number of solved problems as well as the average runtime for successful solutions are given. Since our machine is about two times faster than the hardware used for CASC-17, we doubled all proof times of leanCoP and used a time limit of 250 seconds instead of the 500 seconds used in the competition. leanCoP would have ranked eighth among nine proof systems.

Table 7: CASC-17 results for MIX class with leanCoP's result added

	E	E-SETHEO	...	BLIKSEM	leanCoP	Otter
Attempted	75	75	...	75	75	75
Solved	57	57	...	18	9	8
Av. Time (sec)	79.31	160.53	...	65.33	83.46	55.86

The MIX class is divided into five categories. One of these categories is the HNE category which contains Horn problems with no equality. Again an extract from the final result summary for the HNE category is shown in Table 8. From the ten problems solved by leanCoP all six problems in the planning domain as well as problem SYN311-1 belong to this category. This would have been a remarkable sixth rank among nine proof systems.

4. Refining the Depth-bounded Search

leanCoP uses a depth-first search strategy to explore the search space. After each extension step the new subgoals are considered first before alternative connections are checked. A depth-bounded search is necessary in order to investigate the whole search space up to a certain depth limit. We used the proof depth, i.e.

Table 8: CASC-17 results for HNE category with leanCoP's result added

	E	...	Vampire	leanCoP	Bliksem	Otter	Scott
Attempted	15	...	15	15	15	15	15
Solved	15	...	10	7	3	1	1
Av. Time (sec)	42.40	...	8.36	102.81	179.70	79.00	205.60

the length of the active path, to bound the search depth and use iterative deepening to obtain completeness. This *path-bounded* strategy considers only proofs with $|Path|<PathLim$ for every active path $Path$ and given path limit $PathLim$.

An *inference-bounded* approach uses the number of inferences to limit the search depth. As pointed out in Letz *et al.* (1994) both bounds have their disadvantages: the path-bounded method does not sufficiently restrict the number of inferences, whereas the inference-bounded strategy does not sufficiently limit the depth of the proof, i.e. the length of the active path. A combination of both approaches seems to be an appropriate compromise.

We want to integrate a "lean" combined *path- and inference-bounded* search strategy into leanCoP. For the number of inferences we will only count extension steps and weight each extension step with the number of new subgoal literals contained in the new clause. Let $Path$ be the active path, n the number of extension steps, and $c_1, .., c_n$ the clauses to which a connection step during the proof search has been established. Then we will restrict the proofs to those with

$$|Path| + \sum_{i=1}^{n}(|c_i| - 1) \;<\; Limit \tag{1}$$

where $Limit$ is the depth bound which is used for the iterative deepening search.

4.1. The leanCoP$_i$ Program

We will shortly explain the new version leanCoP$_i$ of our prover realizing the path- and inference-bounded proof search approach. Only minor changes of the Prolog source code were necessary. The Prolog predicate

```
prove_i(Mat,Limit)
```

succeeds if there is a connection proof for the clause set `Mat` of a formula F, which fulfills equation (1). The first proof step where a positive start clause is selected remains unchanged; only a fifth argument is added when calling the actual proof search predicate `prove_i/5`.

```
prove_i(Mat,Limit) :-
     append(MatA,[Cla|MatB],Mat), \+member(-_,Cla),
     append(MatA,MatB,Mat1),
     prove_i([!],[[-!|Cla]|Mat1],[],Limit,_).
```

The Prolog predicate

```
prove_i(Cla,Mat,Path,Limit,Limit1)
```

succeeds if there is a proof for the clause of open subgoals `Cla` using the clauses in `Mat` and the active `Path` which fulfills equation (1). The updated depth bound `Limit1` is returned. The first clause of `prove_i/5` which succeeds for an empty set of open subgoals remains unchanged. The added fifth argument is bound to `Limit` since the proof depth and the number of inferences does not change.

```
prove_i([],_,_,Limit,Limit).
```

The second clause of `prove_i/5` is slightly modified to check the refined depth-bounded condition expressed in equation (1). Instead of calculating the term on the left side of this equation we will subtract $|c_i|-1$ from *Limit* after each extension step and use the updated *Limit* to continue the search. In case of a reduction step the new `Limit3` does not change, i.e. we only add "`Limit3 is Limit`". In case of an extension step we have to add "`length(Cla3,N), Limit2 is Limit-N`". `Cla3` is the clause used for the extension step without the "connection literal" `NegLit` and `Limit2` is the new limit. A fifth argument has to be added for the call of `prove_i/5` to prove the remaining subgoals. Furthermore we move the check $|Path|<Limit$, i.e. "`length(Path,K), K<Limit`" to the beginning of the Prolog clause. This last modification turned out to be more efficient when the refined depth-bounded search strategy is used.

```
prove_i([Lit|Cla],Mat,Path,Limit,Limit1) :-
     length(Path,K), K<Limit,
     (-NegLit=Lit;-Lit=NegLit) ->
        ( member(NegLit,Path), Limit3 is Limit;
          append(MatA,[Cla1|MatB],Mat), copy_term(Cla1,Cla2),
          append(ClaA,[NegLit|ClaB],Cla2), append(ClaA,ClaB,Cla3),
          ( Cla1==Cla2 -> append(MatB,MatA,Mat1);
                          append(MatB,[Cla1|MatA],Mat1)
          ), length(Cla3,N), Limit2 is Limit-N,
          prove_i(Cla3,Mat1,[Lit|Path],Limit2,Limit3)
        ), prove_i(Cla,Mat,Path,Limit3,Limit1).
```

In the original leanCoP program the depth limit is only checked for first-order clauses making an increase of the depth limit for variable-free problems not necessary. Since the leanCoP$_i$ version restricts the depth limit also for variable-free clauses, we have to perform iterative deepening also for variable-free problems. This will slightly change the last Prolog clause which realizes iterative deepening. Note that leanCoP$_i$ is not a decision procedure for propositional logic anymore.

```
prove_i(Mat,Limit) :-
     Limit1 is Limit+1, prove_i(Mat,Limit1).
```

4.2. Performance of leanCoP$_i$

We have tested leanCoP$_i$ on all relevant problems in the TPTP library. The selected problems and the test environment are the same as described in Section 3. leanCoP$_i$ solves 790 (or 35.9%) of the tested 2200 problems, 22 are rated "?" and 57 of them are rated higher than 0.0. All 21 of those problems rated higher than 0.0 which are not already solved by leanCoP are compiled in Table 9 (times are given in seconds). Even though leanCoP$_i$ proves more problems than leanCoP, it is in general a bit slower.

Table 9: Problems rated greater than 0.0 solved by leanCoP$_i$ but not by leanCoP

Problem	Rating	OTTER	PTTP	leanCoP$_i$
CAT001-3	(0.11)	1	>300	19.17
CAT002-3	(0.11)	52	>300	32.92
CAT002-4	(0.17)	2	2.03	11.36
CAT004-4	(0.17)	88	84.72	94.85
CAT012-3	(0.11)	1	7.10	11.35
FLD002-3	(0.67)	1	0.87	283.47
FLD013-4	(0.33)	3	1.09	189.50
FLD016-3	(0.33)	24	0.32	176.94
FLD028-3	(0.33)	25	1.71	197.55
FLD067-3	(0.33)	32	0.14	17.34
GEO058-3	(0.22)	1	0.30	45.06
GEO059-3	(0.22)	>300	0.37	26.59
GEO064-3	(0.12)	1	0.59	68.14
GEO065-3	(0.12)	1	0.56	68.28
GEO066-3	(0.12)	1	0.56	68.37
HEN007-6	(0.17)	1	3.73	155.08
LCL064-1	(0.40)	42	0.81	121.26
LCL230-1	(0.40)	sos-empty	3.73	110.91
LCL231-1	(0.40)	sos-empty	5.43	147.44
SET196-6	(0.12)	17	0.69	142.46
SET197-6	(0.12)	17	0.67	142.59

Table 5 shows how many problems leanCoP$_i$ solves with respect to the problem domain. The results are a bit closer to the results of the PTTP prover which also uses an inference-bounded search. It performs considerably better on domains where PTTP performs well, e.g. the FLD, GEO or HEN domain. On the other hand it performs not so good on the PLA domain on which leanCoP's performance is excellent. The last column of Table 5 considers all problems which are proven by either leanCoP or leanCoP$_i$.

5. Proving Completeness and Correctness

In order to prove completeness and correctness of leanCoP we express the connection calculus by a first-order formula so that $prove(M)$ is a logical consequence of this formula iff there is a derivation for the set of clauses M in the connection calculus. This formula is then translated into a purely declarative Prolog program. We finally show that Prolog's depth-first search is complete for the

$$\frac{}{(\{\}, M, P)} \qquad \textit{axiom}$$

$$\frac{(C, M\backslash C, \{\})}{M} \quad \text{for some positive } C\in M \qquad \textit{start rule}$$

$$\frac{(C\backslash L, M, P)}{(C, M, P)} \quad \begin{array}{l}\text{for some } L\in C, \overline{L}\in P \\ \text{with } (L, \overline{L}) \text{ complementary}\end{array} \qquad \textit{reduction rule}$$

$$\frac{(C\backslash L, M, P) \qquad (C_1\backslash \overline{L}, M\backslash C_1, P\cup\{L\})}{(C, M, P)} \quad \begin{array}{l}\text{for some } L\in C, C_1\in M, \overline{L}\in C_1 \\ \text{with } (L, \overline{L}) \text{ complementary}\end{array} \qquad \textit{extension rule}$$

Figure 1: The connection calculus for propositional logic

constructed Prolog program. We will first concentrate on the propositional case and extend our approach to the first-order case afterwards.

5.1. Propositional Logic

The connection calculus is based on the *matrix characterization* (Bibel, 1987) of logical validity. Basic element is the connection, a pair of literals $(L, \overline{L})$ with the same predicate symbol but with different signs, i.e. one literal contains a negation, the other does not. A pair $(L, \overline{L})$ of *propositional* literals is *complementary* iff they form a connection.

Definition 5.1: *Let M be a matrix, i.e. a set of clauses, and C, C_1 be clauses, i.e. sets of literals. Let $L, \overline{L}$ be literals and P be a path, i.e. a set of literals. The axiom and the rules of the propositional* connection calculus *are given in Figure 1. A matrix M is* provable *iff there is a derivation for M in the connection calculus whose leaves are axioms. (C, M, P) is* provable *iff there is a derivation for (C, M, P) in the connection calculus whose leaves are* axioms.

Lemma 5.1: *A propositional formula F is valid, iff the matrix M of F is provable in the propositional connection calculus.*

Proof: See Bibel (1987). □

Each axiom or rule of the form $\frac{premise}{conclusion}$ is translated into an implication $\forall \ldots [prove(conclusion) \Leftarrow \exists \ldots prove(premise)]$ whereas an empty premise is translated into *true*.

Definition 5.2: *Let positive(C) be true iff the clause C is positive, i.e. does not contain any negation, and compl$(L, \overline{L})$ be true iff the pair $(L, \overline{L})$ is complementary. Let CoCalc be the following first-order formula which expresses the axiom and the rules of the connection calculus in Figure 1.*

$$\forall M, P\ [prove(\{\}, M, P) \Leftarrow true] \qquad \textit{(axiom)}$$
$$\wedge$$
$$\forall M\ [prove(M) \Leftarrow \exists C{\in}M\ (positive(C) \wedge prove(C, M\backslash C, \{\})\ )\ ] \qquad \textit{(start rule)}$$
$$\wedge$$
$$\forall C, M, P\ [prove(C, M, P) \Leftarrow \exists L{\in}C\ \exists \overline{L}{\in}P \qquad \textit{(reduction rule)}$$
$$(compl(L, \overline{L}) \wedge prove(C\backslash L, M, P)\ )\ ]$$
$$\wedge$$
$$\forall C, M, P\ [prove(C, M, P) \Leftarrow \exists L{\in}C\ \exists C_1{\in}M\ \exists \overline{L}{\in}C_1 \qquad \textit{(extension rule)}$$
$$(compl(L, \overline{L}) \wedge prove(C\backslash L, M, P)$$
$$\wedge\ prove(C_1\backslash \overline{L}, M\backslash C_1, P{\cup}\{L\})\ )\ ]$$

LEMMA 5.2: *A matrix M is provable iff the formula $CoCalc \Rightarrow prove(M)$ is valid.*

Proof: We show the following: M or (C, M, P) is provable iff there is a proof for $CoCalc{\vdash}\, prove(M)$ or $CoCalc{\vdash}\, prove(C, M, P)$, respectively, in the sequent calculus LK (Gentzen, 1935). Let M be a matrix, P be a path, C, C_1 be clauses, and $L, \overline{L}$ be literals. Let $\text{Axiom}_{M,P}$ *be the following derivation in LK*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{CoCalc \vdash true}\ axiom \quad \dfrac{}{prove(\{\}, M, P) \vdash prove(\{\}, M, P)}\ axiom}{CoCalc, true{\Rightarrow}prove(\{\}, M, P) \vdash prove(\{\}, M, P)}\ {\Rightarrow}\textit{-left}}{CoCalc, \forall M', P'[true{\Rightarrow}prove(\{\}, M', P')] \vdash prove(\{\}, M, P)}\ \forall\textit{-left}}{CoCalc \vdash prove(\{\}, M, P)}\ \textit{contract-left}$$

and let $\text{Start}_{M,C}$ *be the following derivation in LK:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{CoCalc \vdash prove(C, M, \{\})}}{CoCalc \vdash positive(C) \wedge prove(C, M, \{\})}\ *}{CoCalc \vdash \exists C'(positive(C') \wedge prove(C', M, \{\}))}\ \exists\textit{-right} \quad \dfrac{}{prove(M) \vdash prove(M)}\ axiom}{CoCalc, \exists C'(positive(C') \wedge prove(C', M, \{\})) \Rightarrow prove(M) \vdash prove(M)}\ {\Rightarrow}\textit{-left}}{CoCalc, \forall M'[\exists C'(positive(C') \wedge prove(C', M', \{\})) \Rightarrow prove(M')] \vdash prove(M)}\ \forall\textit{-left}}{CoCalc \vdash prove(M)}\ \textit{contract-left}$$

There are similar derivations for $\text{Reduction}_{M,P,C,L,\overline{L}}$

$$\dfrac{\dfrac{\dfrac{\dfrac{CoCalc \vdash prove(C\backslash L, M, P)}{CoCalc \vdash compl(L, \overline{L}) \wedge prove(C\backslash L, M, P)}\ **}{\vdots} \quad \dfrac{}{prove(C, M, P) \vdash prove(C, M, P)}\ axiom\ \ldots}{\vdots}}{CoCalc \vdash prove(C, M, P)}\ \textit{contract-left}$$

and for $\text{Extension}_{M,P,C,C_1,L,\overline{L}}$ *in LK:*

$$\dfrac{\dfrac{\dfrac{CoCalc \vdash prove(C\backslash L, M, P) \quad CoCalc \vdash prove(C_1\backslash\overline{L}, M\backslash C_1, P\cup\{L\})}{CoCalc \vdash prove(C\backslash L, M, P) \wedge prove(C_1\backslash\overline{L} \wedge M\backslash C_1, P\cup\{L\})}\ \wedge\text{-}right}{CoCalc \vdash compl(L,\overline{L}) \wedge prove(C\backslash L, M, P) \wedge prove(C_1\backslash\overline{L}, M\backslash C_1, P\cup\{L\})}\ ** \quad \cdots \quad \dfrac{}{\vdots}\ axiom}{\vdots} \quad \cdots$$

$$\dfrac{\vdots}{CoCalc \vdash prove(C, M, P)}\ contract\text{-}left$$

"⇒": The proof is by structural induction on the construction of a proof $\mathcal{S}$ for M or (C, M, P) in the connection calculus. Axiom: If the proof S consists only of the axiom-rule then $C{=}\{\}$ and $\text{Axiom}_{M,P}$ *is a proof for $CoCalc \vdash prove(\{\}, M, P)$ in LK. Rules: Let $\mathcal{S}$ be a proof for M or (C, M, P) where the start, reduction or extension rule is the last rule in $\mathcal{S}$, i.e. $\mathcal{S}$ has one of the following forms:*

$$\dfrac{\dfrac{\mathcal{S}_1}{(C,M\backslash C,\{\})}}{M}\ start \qquad \dfrac{\dfrac{\mathcal{S}_2}{(C\backslash L,M,P)}}{(C,M,P)}\ reduction \qquad \dfrac{\dfrac{\mathcal{S}_3}{(C\backslash L,M,P)} \quad \dfrac{\mathcal{S}_4}{(C_1\backslash\overline{L},M\backslash C_1,P\cup\{L\})}}{(C,M,P)}\ extension$$

According to the induction hypothesis there are derivations $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$, and $\mathcal{T}_4$ in LK so that

$$\dfrac{\dfrac{\mathcal{T}_1}{CoCalc \vdash prove(C, M, \{\})}}{CoCalc \vdash prove(M)}\ \text{Start}_{M,C} \qquad \dfrac{\dfrac{\mathcal{T}_2}{CoCalc \vdash prove(C\backslash L, M, P)}}{CoCalc \vdash prove(C, M, P)}\ \text{Reduction}_{M,P,C,L,\overline{L}}$$

$$\dfrac{\dfrac{\mathcal{T}_3}{CoCalc \vdash prove(C\backslash L, M, P)} \quad \dfrac{\mathcal{T}_4}{CoCalc \vdash prove(C_1\backslash\overline{L}, M\backslash C_1, P\cup\{L\})}}{CoCalc \vdash prove(C, M, P)}\ \text{Extension}_{M,P,C,C_1,L,\overline{L}}$$

are proofs for $CoCalc \vdash prove(M)$ or $CoCalc \vdash prove(C, M, P)$, respectively, in the sequent calculus LK. C is a positive clause in $\mathcal{S}_1$, $(L, \overline{L})$ is complementary in $\mathcal{S}_2$ for some $\overline{L}{\in}P$, and $(L, \overline{L})$ is complementary in $\mathcal{S}_3/\mathcal{S}_4$. Therefore positive$(C)$ in $\text{Start}_{M,C}$ *and compl$(L, \overline{L})$ in* $\text{Reduction}_{M,P,C,L,\overline{L}}$ *and* $\text{Extension}_{M,P,C,C_1,L,\overline{L}}$ *are true, and the inferences * and ** are correct.*

"⇐": Every proof for $CoCalc \vdash prove(M)$ or $CoCalc \vdash prove(C, M, P)$ in LK can be build up only by using the derivations $\text{Axiom}_{M,P}$, $\text{Start}_{M,C}$, $\text{Reduction}_{M,P,C,L,\overline{L}}$, *and* $\text{Extension}_{M,P,C,C_1,L,\overline{L}}$. *By structural induction on the construction of such a proof in LK a proof of M or (C, M, P), respectively, in the connection calculus can be constructed.* □

The third and fourth implication of the formula *CoCalc* can be simplified which yields the following equivalent formula

$$\begin{aligned}
&\forall M, P\ [prove(\{\}, M, P)] \\
\wedge\ &\forall M\ [prove(M) \Leftarrow \exists C{\in}M\ (positive(C) \wedge prove(C, M\backslash C, \{\}))] \\
\wedge\ &\forall C, M, P\ [prove(C, M, P) \Leftarrow \exists L{\in}C\ \exists\overline{L}\ (compl(L,\overline{L}) \wedge (\overline{L}{\in}P \vee \exists C_1{\in}M\ \exists\overline{L}{\in}C_1 \\
&\quad prove(C_1\backslash\overline{L}, M\backslash C_1, P\cup\{L\})) \wedge prove(C\backslash L, M, P))]
\end{aligned}$$

```
prove(Mat) :-
     append(MatA,[Cla|MatB],Mat), append(MatA,MatB,Mat1),
     \+member(-_,Cla),
     prove(Cla,Mat1,[]).

prove([],_,_).

prove([Lit|Cla],Mat,Path) :-
     (-NegLit=Lit;-NegLit\=Lit,-Lit=NegLit),
        ( member(NegLit,Path);
          append(MatA,[Cla1|MatB],Mat), append(MatA,MatB,Mat1),
          append(ClaA,[NegLit|ClaB],Cla1), append(ClaA,ClaB,Cla3),
          prove(Cla3,Mat1,[Lit|Path])
        ), prove(Cla,Mat,Path).
```

Figure 2: A declarative version of leanCoP for propositional logic

which can again be transformed into the equivalent formula *CoCalc**:

$$\begin{array}{l} \forall M\ [prove(M) \Leftarrow \exists C{\in}M\ (M_1{=}M\backslash C \wedge positive(C) \wedge prove(C, M_1, \{\})\ )\ ] \\ \wedge\ \forall M, P\ [prove(\{\}, M, P)] \\ \wedge\ \forall C, M, P\ [prove(C, M, P) \Leftarrow \exists L{\in}C\ \exists \overline{L}\ (compl(L, \overline{L}) \wedge (\overline{L}{\in}P \vee \exists C_1{\in}M\ (\overline{L}{\in}C_1 \\ \quad \wedge M_1{=}M\backslash C_1 \wedge C_3{=}C_1\backslash \overline{L} \wedge prove(C_3, M_1, \{L\}{\cup}P)\ )\ ) \wedge prove(C\backslash L, M, P)\ )\ ] \end{array}$$

Lemma 5.3: *The formula CoCalc* ⇒ prove(M) is valid for the matrix M iff the Prolog program in Figure 2 succeeds for the goal* `prove(M)`.

Proof: The formula CoCalc and the Prolog program in Figure 2 are indeed equivalent, since the following propositions hold.*

1. *Implication "⇐", disjunction "∨", and conjunction "∧" are expressed in Prolog by "*`:-`*", "*`;`*", and "*`,`*", respectively. All variables occuring in the head of a Prolog clause are implicitly quantified by universal quantifiers.*
2. *We can consider sets of literals and sets of clauses as ordered multisets. Ordered multisets can be expressed by Prolog lists.*
3. *$\exists X{\in}S\ (S_1{=}X\backslash S \wedge q(X, S_1, S))$ is true iff the Prolog goal "*`append(A,[X|B],S), append(A,B,S1), q(X,S1,S)`*" succeeds. X, S, S_1 correspond to* `X`*,* `S`*,* `S1`*, and* `A`*,* `B` *are fresh variables not occuring elsewhere.*
4. *$positive(C)$ is true iff the goal "*`\+member(-_,C)`*" succeeds.*
5. *$compl(L, \overline{L})$ is true iff "*`-NegLit=Lit;-NegLit\=Lit,-Lit=NegLit`*" succeeds.*
6. *$\exists X{\in}S$ is true iff the goal "*`member(X,S)`*" succeeds.*

7. $q(S) \Leftrightarrow \exists X \in S\,(\,r(X) \wedge q(S \backslash X)\,)$ *is equivalent to* $q(\{X_f\} \cup S_f) \Leftrightarrow (\,r(X_f) \wedge q(S_f)\,)$ *where* X_f *is the first element of the (ordered) set* $\{X_f\} \cup S_f$ *and* S *is a non-empty set.*

We assume the Prolog system to be correct and that sound unification is switched on.¶ *According to the semantics of a Prolog program P the following hold: if* `goal(...)` *succeeds then* $P \Rightarrow goal(...)$ *is valid. If* $P \Rightarrow goal(...)$ *is valid and* `goal(...)` *terminates, then* `goal(...)` *will succeed. Note that the termination condition is essential, since Prolog uses an incomplete depth-first search. Therefore our lemma is true, if the Prolog program in Figure 2 terminates for every goal "*`prove(M)`*" and matrix* `M`.

`append` *as well as* `member` *terminate for all inputs. Therefore* `prove(M)` *terminates for every matrix* `M` *if* `prove(C,M,P)` *terminates for every clause* `C`*, matrix* `M`*, and path* `P`*. The first clause of* `prove(C,M,P)` *always terminates. Let* $\#(C, M, P) := |C| + |M|$ *be the size of a goal* `prove(C,M,P)` *where* $|M|$ *is defined as* $|M| := \sum_{c \in M} |c|$*. Then the sizes of the two* `prove` *goals within the second clause of* `prove(C,M,P)` *are* $\#(C_1 \backslash \overline{L}, M \backslash C_1, P) = |C_1| - 1 + |M| - |C_1| = |M| - 1 < |C| + |M|$ *and* $\#(C \backslash L, M, P) = |C| - 1 + |M| < |C| + |M|$*. Since the size of these goals decreases for each call and this size is always non-negative, i.e.* $\#(C,M,P) \geq 0$ *for all* C, M, P*, every goal* `prove(C,M,P)` *terminates.* □

THEOREM 5.1: *Let F be a (propositional) formula and* `M` *its matrix. The formula F is valid iff* `prove(M)` *succeeds for the Prolog program in Figure 2.*

Proof: Follows immediately from Lemma 5.1, Lemma 5.2, Lemma 5.3, and the equivalence of CoCalc and CoCalc**.* □

5.2. First-order Logic

The approach used for the propositional logic can easily be extended to prove completeness and correctness in the first-order case. Two more concepts have to be integrated into the calculus: appropriate clauses of the given matrix have to be copied and the search depth has to be limited to achieve completeness within Prolog's incomplete search strategy. Like for propositional logic the connection calculus for first-order logic is based on complementary connections. A connection $(\sigma(L), \sigma(\overline{L}))$ of first-order literals is *complementary* under a (first-order) substitution σ iff their arguments are identical under σ.

DEFINITION 5.3: *Let M be a matrix,* C, C_1, C_2 *be clauses,* $L, \overline{L}$ *be literals, P be a path, and* σ *be a substitution. The axiom and the rules of the connection calculus for first-order logic are given in Figure 3. The extension rule is splitted into two versions: the usual one for variable-free clauses C and a new one extension** *for first-order clauses C, i.e. clauses which contain variables. A matrix M is* provable *iff there is a substitution* σ*, a derivation for M in the connection calculus whose leaves are axioms, and all connections are complementary under* σ*.*

$$\frac{}{(\{\}, M, P)} \qquad \textit{axiom}$$

$$\frac{(C, M, \{\})}{M} \quad \text{for some positive } C{\in}M \qquad \textit{start rule}$$

$$\frac{(C\backslash L, M, P)}{(C, M, P)} \quad \begin{array}{l}\text{for some } L{\in}C, \overline{L}{\in}P \\ \text{with } (\sigma(L), \sigma(\overline{L})) \text{ complementary}\end{array} \qquad \textit{reduction rule}$$

$$\frac{(C\backslash L, M, P) \quad (C_1\backslash\overline{L}, M\backslash C_1, P\cup\{L\})}{(C, M, P)} \quad \begin{array}{l}\text{for some } L{\in}C, C_1{\in}M, \overline{L}{\in}C_1 \\ \text{with } (\sigma(L), \sigma(\overline{L})) \text{ complem.}\end{array} \qquad \textit{extension rule}$$

$$\frac{(C\backslash L, M, P) \quad (C_2\backslash\overline{L}, M, P\cup\{L\})}{(C, M, P)} \quad \begin{array}{l}\text{for some } L{\in}C, C_1{\in}M, \overline{L}{\in}C_2 \\ \text{with } (\sigma(L), \sigma(\overline{L})) \text{ complem.} \\ \text{and } C_2 \text{ is a copy of } C_1\end{array} \qquad \textit{extension}^* \textit{ rule}$$

Figure 3: The connection calculus for first-order logic

The calculus slightly differs from the one presented in Bibel (1987) in the way copies of clauses are made.

LEMMA 5.4: *A (first-order) formula F is valid, iff the matrix M of F is provable in the first-order connection calculus.*

Proof: See Bibel (1987). □

DEFINITION 5.4: *Like for the propositional case we can transform the first-order calculus into a formula $CoCalc_{1st}$ (which has already been simplified):*

$$\begin{array}{l} \forall M\ [prove(M) \Leftarrow \exists C{\in}M\ (positive(C) \wedge prove(C, M, \{\}))] \\ \wedge\ \forall M, P\ [prove(\{\}, M, P)] \\ \wedge\ \forall C, M, P\ [prove(C, M, P) \Leftarrow \exists L{\in}C\ \exists\overline{L}\ (compl(\sigma(L), \sigma(\overline{L}))\ \wedge\ (\overline{L}{\in}P\ \vee \\ \quad \exists C_1{\in}M\ \exists C_2\ (copy(C_1, C_2) \wedge \overline{L}{\in}C_2 \wedge C_3{=}C_2\backslash\overline{L} \wedge ((prop(C_2) \wedge M_1{=}M\backslash C_1) \\ \quad \vee (\neg prop(C_2) \wedge M_1{=}M)) \wedge prove(C_3, M_1, \{L\}\cup P))) \wedge prove(C\backslash L, M, P))] \end{array}$$

where $copy(C_1, C_2)$ succeeds iff the clause C_2 is a copy of C_1 where all variables in C_2 have been renamed. $prop(C_2)$ succeeds iff C_2 is a propositional or variable-free clause.

LEMMA 5.5: *A matrix M is provable iff there is a substitution σ so that the formula $CoCalc_{1st} \Rightarrow prove(M)$ is valid.*

Proof: The proof is similar to the propositional case, i.e. we show by structural induction that there is a proof for M in the connection calculus iff there is a proof for $CoCalc_{1st} \vdash prove(M)$ in the sequent calculus LK under σ. □

¶Assuming sound unification for propositional logic is not necessary, but simplifies the proof.

LEMMA 5.6: *The formula* $CoCalc_{1st} \Rightarrow prove(M)$ *is valid for the matrix* M *and some substitution* σ *iff* `prove(M)` *succeeds for the* leanCoP *program shown in Section 2.*

Proof: The formula $CoCalc_{1st}$ *is equivalent to* leanCoP *without the added arguments/predicates to restrict the search depth. In addition to the propositions given in the proof of Lemma 5.3 the following hold:*

1. $prove(C, M, \{\})$ *is true iff the goal "*`prove([!],[[-!|C]|M1],[])`*" succeeds with* $C \in M$ *and* $M_1 = M \setminus C$*. The start step implemented in Prolog uses a variable-free start clause* C *only once which will reduce the search space.*
2. $copy(C_1, C_2)$ *is true iff the goal "*`copy_term(C1,C2)`*" succeeds.*
3. $copy(C_1, C_2) \wedge ((prop(C_2), q(...)) \vee (\neg prop(C_2), r(...)))$ *is true iff* `copy_term(C1,C2),(C1==C2 -> q(...) ; r(...))` *succeeds.*
4. *The substitution* σ *is calculated implicitly by Prolog.*
5. *Predicates within a (declarative) Prolog program can be reordered.*
6. *The goal "*`(-NegLit=Lit;-NegLit\=Lit,-Lit=NegLit), ...`*" succeeds iff "*`(-NegLit=Lit;-Lit=NegLit) -> ...`*" (which contains an implicit cut) succeeds.*

Sound unification has to be used in Prolog. Finally we show that Prolog's depth-first search is complete for the leanCoP *program:* `prove(M,I)` *terminates for every matrix* `M` *and path limit* `I`*. Similar to the propositional case we define the size of* `prove(C,M,P,I)` *as a tuple, i.e.* $\#(C, M, P, I) := (|C|+|M|, |P|)$ *with* $|M| := \sum_{c \in M} |c|$*. For each call the first element of* $\#(C, M, P, I)$ *decreases or the second one increases. Whenever the first element does not decrease, i.e.* $|P|$ *is increased, it is checked whether* $|P|$ *is smaller than the given path limit* I*. Since* $|C|+|M|$ *is non-negative, every goal* `prove(C,M,P,I)` *terminates and therefore* `prove(M,I)` *terminates. Performing iterative deepening on* `I` *yields completeness for the first-order case.* □

THEOREM 5.2: *Let* F *be a formula and* `M` *its matrix. The formula* F *is valid iff* `prove(M)` *succeeds for the Prolog program* leanCoP *shown in Section 2.*

Proof: Follows immediately from Lemma 5.4, Lemma 5.5, and Lemma 5.6. □

6. Conclusion, Related Work and Outlook

We have presented a compact Prolog theorem prover for first-order (clause) logic which implements the basic connection calculus. It is sound, complete, and a decision procedure for propositional logic. Due to the compact code the program can easily be modified for special purposes or applications. On the other hand the Prolog program gives a short declarative description of the connection calculus. The goal-oriented approach yields an astonishing performance, in particular for

Horn problems without equality. We ran leanCoP on a subset of the TPTP library and compared its performance with the resolution-based prover OTTER, the compilation-based prover PTTP, and the tableau-based prover lean*TAP*. Even though the performance of OTTER, a much larger and sophisticated system, is in general better, leanCoP is able to solve several difficult problems for which OTTER does not find a proof. PTTP is a much smaller implementation, though the source code (including comments) still fills about 18 pages. It translates a given set of clauses into a Prolog program and then uses Prolog's inference system to carry out the actual proof search. This yields an inference rate which is an order of magnitude higher than the inference rate achieved with leanCoP. Still leanCoP and the refined version leanCoP$_i$ are able to solve almost as many problems from the TPTP library as PTTP does. lean*TAP*'s source code has a size very similar to the size of leanCoP, but behaves rather poor on problems in clausal form. For problems in non-clausal form lean*TAP*'s performance is expected to be much closer to that of leanCoP. We integrated a combined path- and inference-bounded search into leanCoP which improves its behaviour on the TPTP library. Finally we proved completeness and correctness by stepwisely transforming the connection calculus into an equivalent declarative Prolog program.

Even though leanCoP is able to solve hard problems from the TPTP library, it is not intended to be a state-of-the-art prover. To solve e.g. difficult mathematical problems, theorem provers like OTTER or E-SETHEO (Stenz and Wolf, 2000) are more appropriate. But for a lot of applications state-of-the-art performance is not required. For example for interactive proof editors the integration of fully automatic provers can assist humans to find proofs. Lean provers can easily be integrated and modified by people who do not have a deep knowledge about fully automatic provers. Since it is much easier (and faster) to understand a few lines of Prolog code than several thousand lines of e.g. C code, lean theorem provers are also very well suited for teaching purposes. Finally for the same reason it is also much easier to verify completeness and correctness of lean theorem provers.

In Neugebauer and Schaub (1991) a *pool-based* connection calculus together with an one-page Prolog program is described. Though the underlying calculus is similar, the actual implementation technique is different. Furthermore the positive-start-clause technique as well as the restriction of clause copies to first-order clauses are missing. In contrast to leanCoP it is not a decision procedure for propositional formulas. Another lean prover for classical logic is SATCHMO (Manthey and Bry, 1988) which is a short model-generation prover written in Prolog. Input clauses are modified within the Prolog database making an extensive use of `assert` and `retract` necessary, which destroys the declarative semantics of the Prolog program. SATCHMO does essentially ground level reasoning and performs rather poor on the problems in the TPTP library.

Due to its compact size new techniques can easily be integrated into leanCoP's code. This makes experimental evaluations of novel techniques very easy. We have, for example, implemented a slightly modified version of leanCoP where the given set of clauses is stored in Prolog's database (i.e. one Prolog clause for each

literal) instead of representing it as a Prolog list. This technique combines the advantages of "Prolog technology" theorem provers (like e.g. PTTP) and "lean" theorem provers by using Prolog's fast inference machine to find connections without losing readability, modifiability, and flexibility of lean implementations. Experimental results showed that it improves the performance of leanCoP considerably (e.g. SET016-7 from Table 3 is proved in 1.87 seconds instead of 183.31 seconds). On an average the timings for solving problems of the TPTP library are about ten times faster. Other possible improvements include the integration of factorization, lemmata or the folding up rule (Letz *et al.*, 1994) as well as avoiding the use of contrapositives (Baumgartner and Furbach, 1994).

We have also implemented a lean *non-clausal* version of leanCoP for propositional logic. It does not need the input formula to be in clausal form but preserves its structure throughout the entire proof search, thus combining the advantages of non-clausal tableau calculi and goal-oriented connection-based provers. The extension to first-order logic though needs some efforts, since copying of appropriate subformulas cannot be done so easily in a lean way. A non-clausal connection-based prover can also be extended to some non-classical logics, like intuitionistic, modal or linear logic (Otten and Kreitz, 1996; Kreitz and Otten, 1999). We only have to add an additional prefix unification procedure (Otten and Kreitz, 1996) leaving the actual proof search procedure unchanged. Similar approaches using labels or prefixes have already been used to implement lean provers based on free-variable semantic tableaux for intuitionistic logic (Otten, 1997), modal logics (Beckert and Goré, 1997), and linear logic (Mantel and Otten, 1999). Thus leanCoP can serve as a basis for lean connection-based theorem provers for logics for which up to now only lean tableau-based provers have been realized.

The source code of leanCoP together with more information can be found at `http://www.leancop.de` .

Acknowledgements

The authors would like to thank the referees for their useful comments.

References

Argonne National Laboratory, Mathematics and Computer Science Division. OTTER and MACE on TPTP v2.3.0. `http://www-unix.mcs.anl.gov/AR/otter/tptp230.html`, 2000.

O. Astrachan, D. Loveland. METEORs: High performance theorem provers using model elimination. In R. Boyer, ed., *Automated Reasoning: Essays in Honour of Woody Bledsoe*. Kluwer, 1991.

P. Baumgartner, U. Furbach. Model elimination without contrapositives. *12^{th} CADE*, LNAI 814, pp. 87–101. Springer, 1994.

B. Beckert, R. Goré. Free variable tableaux for propositional modal logics. *TABLEAUX '97*, LNAI 1227, pp. 91–106. Springer, 1997.

B. Beckert, J. Posegga. lean*TAP*: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15:339–358. Kluwer, 1995.

W. Bibel. Matings in matrices. *Communications of the ACM*, 26:844–852, 1983.

W. Bibel. *Automated Theorem Proving*. Vieweg, second edition, 1987.

W. Bibel. *Deduction: Automated Logic*. Academic Press, 1993.

W. Bibel, S. Brüning, U. Egly, T. Rath. KoMeT. *12^{th} CADE*, LNAI 814, pp. 783–787. Springer, 1994.

W. Clocksin, C. Mellish, *Programming in Prolog*. Springer, 1981.

G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

C. Kreitz, J. Otten. Connection-based theorem proving in classical and non-classical logics. *Journal of Universal Computer Science*, 5:88–112. Springer, 1999.

R. Letz, J. Schumann, S. Bayerl, W. Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212. Kluwer, 1992.

R. Letz, K. Mayr, C. Goller. Controlled integration of the cut rule into connection tableaux calculi. *Journal of Automated Reasoning*, 13:297–337. Kluwer, 1994.

D. Loveland. Mechanical theorem proving by model elimination. *Journal of the ACM*, 15:236–251, 1968.

H. Mantel, J. Otten. linTAP: A tableau prover for linear logic. *TABLEAUX '99*, LNAI 1617, pp. 217–231. Springer, 1999.

R. Manthey, F. Bry. SATCHMO: A theorem prover implemented in Prolog. *9^{th} CADE*, LNCS 310, pp. 415–434. Springer, 1988.

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

M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, K. Mayr. SETHEO and E-SETHEO – The CADE-13 systems. *Journal of Automated Reasoning*, 18:237–246. Kluwer, 1997.

G. Neugebauer, T. Schaub. A pool-based connection calculus. Technical Report AIDA-91-02, Intellektik, TH Darmstadt, 1991.

J. Otten, C. Kreitz. T-string-unification: Unifying prefixes in non-classical proof methods. *Proc. 5^{th} TABLEAUX Workshop*, LNAI 1071, pp. 244–260. Springer, 1996.

J. Otten, C. Kreitz. A uniform proof procedure for classical and non-classical logics. *KI-96: Advances in Artificial Intelligence*, LNAI 1137, pp. 307–319. Springer, 1996.

J. Otten. leanTAP: An intuitionistic theorem prover. *TABLEAUX '97*, LNAI 1227, pp. 307–312. Springer, 1997.

F. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216. Kluwer, 1986.

J. Posegga, P. Schmitt. Implementing semantic tableaux. In M. D'Agostino, D. Gabbay, R. Hähnle and J. Posegga, eds, *Handbook of Tableau Methods*, pp. 581–629. Kluwer, 1999.

G. Stenz, A. Wolf. E-SETHEO: An automated theorem prover. *TABLEAUX 2000*, LNAI 1847, pp. 436–440. Springer, 1997.

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

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

G. Sutcliffe, C. Suttner. The TPTP problem library - CNF release v1.2.1. *Journal of Automated Reasoning*, 21:177–203. Kluwer, 1998.