leanCoP: Lean Connection-Based Theorem Proving

Jens Otten Wolfgang Bibel

Fachgebiet Intellektik, Fachbereich Informatik
Darmstadt University of Technology
Alexanderstr. 10, 64283 Darmstadt, Germany
{jeotten,bibel}@informatik.tu-darmstadt.de

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, complete (if one more line is added), and demonstrates a comparatively strong performance.

1 Introduction

The connection calculus [4–6], the connection tableau calculus [10] and the similar model elimination calculus [11] are popular and successful proof procedures because of their goal-oriented search strategy. Proof systems based on one of these approaches, e.g. [7,9,15,2] to name a few, have shown to be an appropriate basis to automate formal reasoning in 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. Our prover even finds proofs for 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, as already demonstrated with programs like e.g. leanTAP [3] and SATCHMO [13].

In the rest of this paper we will explain the source code (in a more readable form) and present some performance results achieved with our prover on the problems contained in the TPTP library [18]. The source code of leanCoP can be obtained at http://www.intellektik.informatik.tu-darmstadt.de/~jeotten/leancop .

2 The Program

Our prover is based on the *connection calculus* [4–6] which is a proof procedure for (full) first-order clause logic. It starts by first selecting a *start clause* before *extension steps* and *reduction steps* are repeatedly applied. Whereas the extension step connects a subgoal literal to a negated literal of a new clause, the reduction step connects it to a negated literal of the so-called *active path*. The extension step actually realizes the goal-oriented proof search. This approach can also be considered as constructing a *connection tableaux* [10] where open subgoals are selected in a depth-first way. To prove a formula we first need to translate the given first-order formula into a set of clauses. We use the positive representation, 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 formula $(\exists X(p \Rightarrow f(X)) \wedge \exists X(f(X) \Rightarrow p)) \Rightarrow \exists X(p \Leftrightarrow f(X))$ (problem SYN051-1 in [18]) which can be translated to $(p \wedge \neg f(a)) \vee (f(b) \wedge \neg p) \vee (p \wedge f(X)) \vee (\neg p \wedge \neg f(X))$ represented by the clause set $\{\{p, \neg f(a)\}, \{f(b), \neg p\}, \{p, f(X)\}, \{\neg p, \neg f(X)\}\}$.

We will use Prolog lists to represent sets, Prolog atoms to represent atomic formulas 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)]]`. We use the Prolog predicates `prove/2` and `prove/4` to implement our prover.

`prove(Mat,PathLim)`

succeeds if there is a connection proof for the clause set `Mat`, the so-called *matrix*, of a formula F, whose active path lengths 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` without literals of the form $\neg q$ (i.e. `\+member(-_,Cla)` succeeds). Afterwards the predicate which realizes extension and reduction step is called. We start the proof search with the subgoal containing only the special literal "!" (which should not occur in `Mat`) and add the literal "¬!" (which will be used for the first extension step) to the original start clause. This kind of initial step is essential since otherwise no copies of the start clause are made (which might be necessary in some cases).[1]

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

`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 length is limited by `PathLim`.

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

If the clause of open subgoals is empty, we do not have to perform any further search. 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),
          ( 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. if `NegLit` is an element of `Path`, using the (built-in) predicate `member`. For this sound unification has to be used.[2] 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)[3] 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 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` (which is necessary to achieve completeness).[4] Afterwards `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`.

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

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

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

[4] `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.

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

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.[5] It fails if the set of input clauses `Mat` does not contain any variable, thus yielding a decision procedure for propositional logic.

3 Performance

We have tested leanCoP on all valid (i.e. unsatisfiable) first-order problems in clause form and all propositional problems in clause form contained in the current version 2.3.0 of the TPTP library (see also [18]). No reordering of clauses has been done. When transforming the formulas into an appropriate input format we translated literals of the form `++q` into `-q` and `--q` into `q`, respectively, since we use a positive representation.

The 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 60 seconds.

number of all tested problems	problems solved by leanCoP (within 60 seconds)	
2200 (100%)	667 (30.3%)	386 within 0 to 1 second
		187 within 1 to 10 seconds
		94 within 10 to 60 seconds

Table 1. Overall performance of leanCoP on the TPTP library

Even though a lot of the problems are rather hard, leanCoP was able to solve 667 (=30.3%) problems, 386 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, where 0.0 means that all state-of-the-art provers can solve the problem and 1.0 means that no state-of-the-art prover can solve it. leanCoP was able to solve 48 problems rated higher than 0.0. They are compiled in Table 2. For each of these problems its name and rating is given as well as the timings (in seconds) of OTTER 3.1 (see also [14]) and leanCoP. The timings of OTTER are taken from [1].

Surprisingly there are 31 problems which OTTER cannot solve (">300" or "**s**et **o**f **s**upport empty" in Table 2) within 300 seconds on a 400 MHz Linux machine (which should be slightly faster than our machine). Most of them are within the Field Theory domain (FLD) and the Planning domain (PLA). Altogether OTTER is able to solve 1602 (=72.8%) out of the tested 2200 problems.

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

Problem	Rating	OTTER	leanCoP
BOO012-1	(0.17)	3	30.63
CAT003-2	(0.50)	>300	39.07
CAT003-3	(0.11)	>300	7.42
CAT012-4	(0.17)	1	50.54
FLD013-1	(0.67)	>300	1.49
FLD023-1	(0.33)	>300	1.84
FLD025-1	(0.67)	>300	1.44
FLD030-1	(0.33)	1	0.10
FLD030-2	(0.33)	>300	1.40
FLD037-1	(0.33)	>300	4.96
FLD060-1	(0.67)	>300	1.78
FLD061-1	(0.67)	>300	2.10
FLD067-1	(0.33)	>300	4.43
FLD070-1	(0.33)	>300	7.71
FLD071-3	(0.33)	2	1.13
GRP008-1	(0.22)	1	2.62
LCL045-1	(0.20)	119	2.10
LCL097-1	(0.20)	1	0.87
LCL111-1	(0.20)	1	0.30
LCL130-1	(0.20)	1	0.04
LCL195-1	(0.20)	sos-empty	32.28
NUM283-1.005	(0.20)	1	0.55
PLA004-1	(0.40)	>300	13.59
PLA004-2	(0.40)	>300	20.54

Problem	Rating	OTTER	leanCoP
PLA005-1	(0.40)	>300	1.54
PLA005-2	(0.40)	>300	0.37
PLA007-1	(0.40)	>300	0.50
PLA009-1	(0.40)	>300	0.20
PLA009-2	(0.40)	>300	7.30
PLA011-1	(0.40)	>300	0.51
PLA011-2	(0.40)	>300	1.56
PLA013-1	(0.40)	>300	0.85
PLA014-1	(0.40)	>300	7.20
PLA014-2	(0.40)	>300	7.49
PLA016-1	(0.40)	>300	0.23
PLA019-1	(0.40)	>300	0.22
PLA021-1	(0.40)	>300	0.64
PLA022-1	(0.40)	>300	1.33
PLA022-2	(0.40)	>300	0.09
PUZ034-1.004	(0.67)	sos-empty	40.53
RNG006-2	(0.20)	5	1.00
RNG040-1	(0.11)	1	0.04
RNG040-2	(0.22)	1	0.83
SET060-6	(0.12)	1	0.64
SET060-7	(0.12)	1	0.71
SET152-6	(0.12)	1	48.04
SET153-6	(0.12)	>300	9.91
SYN048-1	(0.20)	1	0.01

Table 2. TPTP problems with ratings greater than 0.0 solved by leanCoP

lean*TAP* [3] only solves 135 (=6.1%) out of the tested 2200 TPTP problems, two of them (FLD067-1, SYN048-1) are rated higher than 0.0. leanCoP solves every problem which is solved by lean*TAP* except problem SYN350-1. Five of the problems in Table 2 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, 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.

4 Conclusion, Related Work and Outlook

We have presented a compact Prolog theorem prover for first-order (clause) logic which implements the basic connection calculus. The goal-oriented approach yields an astonishing performance. We ran leanCoP on a subset of the TPTP library and were able to solve difficult problems for which current state-of-the-art provers do not find a proof. 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.

Other lean provers for classical logic are SATCHMO and lean*TAP*. SATCHMO [13] is a short model-generation prover written in Prolog. Input clauses are kept in the Prolog database making an extensive use of `assert` and `retract` necessary. In its basic version it can only deal with range-restricted clauses. lean*TAP* [3] is a compact Prolog implementation of a free-variable analytic tableau cal-

culus for formulas in negation normal form. Having good performance on non-clausal problems, it behaves rather poor on problems in clausal form.

We have implemented an only slightly longer non-clausal version of our program 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. Unfortunately the extension to first-order logic cannot be done so easily, since copying of appropriate subformulas is a difficult task. A non-clausal prover can also be extended to some non-classical logics, like intuitionistic, modal or linear logic [16,8]. 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 [17,12] have been realized.

References

1. Argonne National Laboratory. Otter and MACE on TPTP v2.3.0. Web page at `http://www-unix.mcs.anl.gov/AR/otter/tptp230.html`, May 2000.
2. O. Astrachan, D. Loveland. METEORs: High performance theorem provers using model elimination. In Boyer, *Automated Reasoning: Essays in Honour of Woody Bledsoe*. Kluwer, 1991.
3. B. Beckert and J. Posegga. leanTAP: lean, tableau-based theorem proving. *12th CADE*, LNAI 814, pp. 793–797, 1994.
4. W. Bibel. Matings in matrices. *Communications of the ACM*, 26:844–852, 1983.
5. W. Bibel. *Automated Theorem Proving*. Vieweg, second edition, 1987.
6. W. Bibel. *Deduction: Automated Logic*. Academic Press, 1993.
7. W. Bibel, S. Brüning, U. Egly, T. Rath. Komet. *12th CADE*, LNAI 814, pp. 783–787, 1994.
8. C. Kreitz and J. Otten. Connection-based theorem proving in classical and non-classical logics. *Journal of Universal Computer Science*, 5:88–112, 1999.
9. R. Letz, J. Schumann, S. Bayerl, W. Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.
10. R. Letz, K. Mayr, C. Goller. Controlled integration of the cut rule into connection tableaux calculi. Journal of Automated Reasoning, 13: 297–337, 1994.
11. D. Loveland. Mechanical theorem proving by model elimination. *JACM*, 15:236–251, 1968.
12. H. Mantel and J. Otten. linTAP: A tableau prover for linear logic. *8th TABLEAUX Conference*, LNAI 1617, pp. 217–231, 1999.
13. R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. *9th CADE*, LNCS 310, pp. 415–434, 1988.
14. W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.
15. 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, 1997.
16. J. Otten and C. Kreitz. A uniform proof procedure for classical and non-classical logics. *KI-96: Advances in Artificial Intelligence*, LNAI 1137, pp. 307–319, 1996.
17. J. Otten. ileanTAP: An intuitionistic theorem prover. *6th TABLEAUX Conference*, LNAI 1227, pp. 307–312, 1997.
18. G. Sutcliffe, C. Suttner. The TPTP problem library - CNF release v1.2.1. *Journal of Automated Reasoning*, 21: 177–203 (1998)