leanT^AP: Lean Tableau-Based Theorem Proving
— Extended Abstract —

Bernhard Beckert & Joachim Posegga

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
76128 Karlsruhe, Germany
{beckert|posegga}@ira.uka.de

Abstract.

```
“prove((E,F),A,B,C,D) :- !, prove(E,[F|A],B,C,D).
 prove((E;F),A,B,C,D) :- !, prove(E,A,B,C,D), prove(F,A,B,C,D).
 prove(all(H,I),A,B,C,D) :- !,
     \+length(C,D), copy_term((H,I,C),(G,F,C)),
     append(A,[all(H,I)],E), prove(F,E,B,[G|C],D).
 prove(A,_,[C|D],_,_) :-
     ((A= -(B); -(A)=B)) -> (unify(B,C); prove(A,[],D,_,_)).
 prove(A,[E|F],B,C,D) :- prove(E,F,[A|B],C,D).”
```

implements a first-order theorem prover based on free-variable semantic tableaux. It is complete, sound, and efficient.

1 Introduction

The Prolog program listed in the abstract implements a complete and sound theorem prover for first-order logic based on free-variable semantic tableaux [6]. We call this *lean theorem proving*: the idea is to achieve maximal efficiency from minimal means. We will see that the above program is indeed very efficient—not *although* but *because* it is extremely short and compact.

Satchmo [7] can be regarded the earliest application of lean theorem proving. The core of Satchmo is about 15 lines of Prolog code, and for implementing a refutation complete version another 15 lines are required. Unfortunately, Satchmo works only for formulæ in clausal form (CNF).

Many problems become much harder when translating them to clausal form, so it seems much better to avoid CNF and to preserve position and scope of quantifiers.[1] One way to achieve this is to use a calculus based on free-variable tableaux. It is a common, but mistaken belief that tableau calculi are inefficient; we will demonstrate the contrary.

The reader is assumed to be familiar with free-variable tableaux (e.g. [6]) and the basics of Prolog. The full version of this paper [4] is available upon request

[1] Using a definitional CNF [5] helps at most partially: it avoids exponential growth of formulæ for the price of introducing some redundancy into the proof search. Extending the scope of quantifiers to clause level, however, cannot be avoided.

from the authors. The source code of lean$T^{A}P$ (and that of a slightly improved version; see Section 4) can also be obtained free of charge.

2 The Program

The idea behind lean$T^{A}P$ is to exploit the power of Prolog's inference engine as much as possible; whilst Satchmo is based upon `assert` and `retract`, we do not use these predicates at all but rely on Prolog's clause indexing scheme and backtracking mechanism. We modify Prolog's depth-first search to bounded depth-first search for gaining a complete prover.

For the sake of simplicity, our considerations are restricted to first-order formulæ in Skolemized negation normal form. This is not a serious restriction; the prover can easily be extended to full first-order logic by adding the standard tableau rules.[2] We will use Prolog syntax for first-order formulæ: atoms are Prolog terms, "`-`" is negation, "`;`" disjunction, and "`,`" conjunction. Universal quantification is expressed as `all(X,F)`, where `X` is a Prolog variable and `F` is the scope. Thus, a first-order formula is represented by a Prolog term (e.g., `(p(0),all(N,(-p(N);p(s(N)))))` stands for $p(0) \wedge (\forall n(\neg p(n) \vee p(s(n))))$).

We use a single Prolog predicate to implement our prover:

```
prove(Fml,UnExp,Lits,FreeV,VarLim)
```

succeeds if there is a closed tableau for the first-order formula bound to `Fml`. The proof proceeds by considering individual branches (from left to right) of a tableau; the parameters `Fml`, `UnExp`, and `Lits` represent the current branch: `Fml` is the formula being expanded, `UnExp` holds a list of formulæ not yet expanded, and `Lits` is a list of the literals present on the current branch. `FreeV` is a list of the free variables on the branch (these are Prolog variables, which might be bound to a term). A positive integer `VarLim` is used to initiate backtracking: it is an upper bound for the length of `FreeV`.

We will briefly go through the program listed in the abstract again (using a more readable form now) and explain its behavior. The prover is started with the goal `prove(Fml,[],[],[],VarLim)`, which succeeds if `Fml` can be proven inconsistent without using more than `VarLim` free variables on each branch.

If a conjunction (α-formula[3]) "A and B" is to be expanded, then "A" is considered first and "B" is put in the list of not yet expanded formulæ:

```
prove((A,B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,[B|UnExp],Lits,FreeV,VarLim).
```

For disjunctions (β-formulæ) we split the current branch and prove two new goals:

[2] Skolemization has to be carried out very carefully, since straightforwardly Skolemizing can easily hinder finding a proof [3]. The full version [4] of this paper gives more details.

[3] Due to R. Smullyan, conjunctive type formulæ are called α-formulæ in the semantic tableaux framework.

```
prove((A;B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,UnExp,Lits,FreeV,VarLim),
    prove(B,UnExp,Lits,FreeV,VarLim).
```

Handling universally quantified formulæ (γ-formulæ) requires a little more effort. We first check the number of free variables on the branch. Backtracking is initiated if the depth bound `VarLim` is reached. Otherwise, we generate a "fresh" instance of the current γ-formula `all(X,Fml)` with `copy_term`. `FreeV` is used to avoid renaming the free variables in `Fml`. The original γ-formula is put at the end of `UnExp`[4], and the proof search is continued with the renamed instance `Fml1` as the formula to be expanded next. The copy of the quantified variable, which is now free, is added to the list `FreeV`:

```
prove(all(X,Fml),UnExp,Lits,FreeV,VarLim) :- !,
    \+ length(FreeV,VarLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[all(X,Fml)],UnExp1),
    prove(Fml1,UnExp1,Lits,[X1|FreeV],VarLim).
```

The next clause closes branches; it is the only one which is not determinate. Note, that it will only be entered with a literal as its first argument. `Neg` is bound to the negated literal and sound unification[5] is tried against the literals on the current branch. The clause calls itself recursively and traverses the list in its second argument; no other clause will match since `UnExp` is set to the empty list for this recursion.

```
prove(Lit,_,[L|Lits],_,_) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L); prove(Lit,[],Lits,_,_)).
```

Note, that the implication "`->`" introduces an implicit cut after binding `Neg`: this prevents generating double negation when backtracking (which would happen, if "`,`" were used instead).

The last clause is reached if the preceding clause cannot close the current branch. We add the current formula (always a literal) to the list of literals on the branch and pick a formula waiting for expansion:

```
prove(Lit,[Next|UnExp],Lits,FreeV,VarLim) :-
    prove(Next,UnExp,[Lit|Lits],FreeV,VarLim).
```

leanT^AP has two choice points: One is selecting between the last two clauses, which means closing a branch or extending it. The second choice point within the fourth clause enumerates closing substitutions during backtracking.

[4] Putting it at the top of the list destroys completeness: the same γ-formula would be used over and over again until the depth bound is reached (bracktracking does not change the order in which formulæ are expanded).

[5] In contrary to the built-in unification `=`, the predicate `unify` implements sound unification, i.e., unification with occurs check.

3 Performance

Although (or better: because) the prover is so small, it shows striking performance. Table 1 shows experimental results for a subset of Pelletier's problems [8].

Table 1. leanT^AP's performance for Pelletier's problem set (the runtime has been measured on a SUN SPARC 10 workstation with SICStus Prolog 2.1; "0 *msec*" means "not measurable").

No.	Limit	Branches	Closings	Time
	`VarLim`	closed	tried	*msec*
17	1	14	14	0
18	2	1	1	9
19	2	4	6	0
20	6	3	3	9
21	2	8	8	0
22	2	7	14	9
23	1	4	4	0
24	6	33	33	9
25	3	5	5	0
26	3	16	17	0
27	4	8	8	0
28	3	5	5	0
29	2	11	11	9
30	2	4	4	9
31	3	5	5	0
32	3	10	10	10
33	1	11	11	0
34	??	–	–	∞
35	4	1	1	0
36	6	3	3	0
37	7	8	8	9
38	4	90	101	210
39	1	2	2	0
40	3	4	5	0
41	3	4	5	0
42	3	5	5	9
43	5	18	18	109
44	3	5	5	10
45	5	17	17	39
46	5	53	63	59

Some of the theorems, like Problem 38, are quite hard: the $_3T^AP$ prover [1], for instance, needs more than ten times as long. If leanT^AP can solve a problem, its performance is in fact comparable to compilation-based systems that search for proofs by generating Prolog programs and running them [10, 9].

Pelletier No. 34 (also called "Andrews Challenge") is a surprise: all others run really fast but for Problem 34 leanT^AP does not find a proof. One reason might be that we did not invest much time in finding the right number of free variables to allow: an iterative deepening approach (as applicable to all other problems) does not work: if `VarLim` is set to 4, the prover does not return after 30 minutes. It is easily possible that the right limit (which is above 12) returns a proof very quickly.

4 Conclusion & Outlook

We showed that a first-order calculus based on free-variable semantic tableaux can be efficiently implemented in Prolog with minimal means.

One could regard lean$T^{A}P$ as a Prolog hack. However, we think it demonstrates more than tricky use of Prolog: it shows that semantic tableaux are an efficient calculus when implemented carefully. Besides this, the philosophy of "lean theorem proving" is interesting: We showed that it is possible to reach considerable performance by using extremely compact (and efficient) code instead of elaborate heuristics. One should not confuse "lean" with "simple": each line of a "lean" prover has to be coded with a lot of careful consideration.

There is still room for improvement without sacrificing simplicity and/or elegance of our approach: lean$T^{A}P$ can easily be extended by techniques known to enhance the performance of tableau-based provers: A slightly longer version making use of "universal formulæ" [2] can, for example, solve Pelletier No. 34 in about $100 msec$ (see [4] for details).

Another possibility is to use an additional preprocessing step that translates a negation normal form into a graphical representation of a fully expanded tableau (see [9] for details). This can be implemented equivalently simply and requires only linear effort at runtime. The prover itself then reduces to two clauses, since no compound formulæ are present any more and all branches are already fully developed. The speedup will not be dramatic, but considerable. Furthermore, we can implement the compilation principle described by Posegga [9]: the idea is to translate tableau graphs into Prolog clauses that carry out the proof search at runtime. Compared with "conventional" implementations of tableau-based systems, this gains about one order of magnitude of speed. It will be subject to future research to apply this principle in the spirit of lean deduction.

References

1. B. Beckert, S. Gerberding, R. Hähnle, and W. Kernig. The tableau-based theorem prover $_3T^{A}P$ for multiple-valued logics. In *Proceedings, CADE 11, Albany/NY*, LNCS. Springer, 1992.
2. B. Beckert and R. Hähnle. An improved method for adding equality to free variable semantic tableaux. In *Proceedings, CADE 11, Albany/NY*, LNCS. Springer, 1992.
3. B. Beckert, R. Hähnle, and P. H. Schmitt. The even more liberalized δ-rule in free variable semantic tableaux. In *Proceedings, 3rd Kurt Gödel Colloquium, Brno, Czech Republic*, LNCS. Springer, 1993.
4. B. Beckert and J. Posegga. lean$T^{A}P$: Lean tableau-based theorem proving. To appear (available upon request from the authors), 1994.
5. E. Eder. *Relative Complexities of First-Order Calculi.* Artificial Intelligence. Vieweg Verlag, 1992.
6. M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 1990.
7. R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In *Proceedings, CADE 9, Argonne*, LNCS. Springer, 1988.
8. F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.
9. J. Posegga. Compiling proof search in semantic tableaux. In *Proceedings, ISMIS 7, Trondheim, Norway*, LNCS. Springer, 1993.
10. M. E. Stickel. A prolog technology theorem prover. In *Proceedings, CADE 9, Argonne*, LNCS. Springer, 1988.