Lean Theorem Proving: Maximal Efficiency from Minimal Means

Bernhard Beckert & Joachim Posegga

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

```
prove((A,B),UnExp,Lits,FreeV,VLim) :- !,
     prove(A,[B|UnExp],Lits,FreeV,VLim).
prove((A;B),UnExp,Lits,FreeV,VLim) :- !,
     prove(A,UnExp,Lits,FreeV,VLim),
     prove(B,UnExp,Lits,FreeV,VLim).
prove(all(X,Fml),UnExp,Lits,FreeV,VLim) :- !,
    \+ length(FreeV,VLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[all(X,Fml)],UnExp1),
    prove(Fml1,UnExp1,Lits,[X1|FreeV],VLim).
prove(Lit,_,[L|Lits],_,_) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L); prove(Lit,[],Lits,_,_)).
prove(Lit,[Next|UnExp],Lits,FreeV,VLim) :-
    prove(Next,UnExp,[Lit|Lits],FreeV,VLim).
```

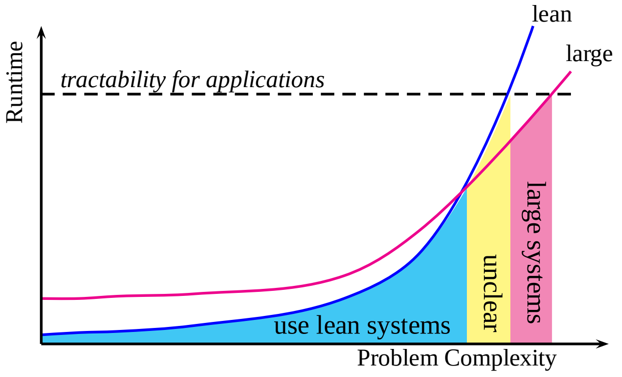
Researchers in Automated Reasoning often complain that there are sparse applications of the techniques they develop. One reason might be that implementation-oriented research favors huge and highly complex systems. It is hard to see how to apply these besides using them as a black box. Adaptability, however, is an important criterion for applying techniques; this discrepancy can be overcome by using *lean* theorem provers.

The idea of *lean theorem proving* is to achieve maximal efficiency from minimal means. Every possible effort is made to eliminate overhead; based on experience in implementing (complex) deduction systems, only the most important and efficient techniques and methods are implemented. Satchmo [4] 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 range-restricted formulæ in clausal form (CNF).

Another instance of lean theorem proving is listed on the upper right. This Prolog program, called leanT^AP [2], implements a complete and sound theorem prover for first-order logic, based on free-variable semantic tableaux [3] (see the appendix for a detailed description). Whilst Satchmo extensively uses `assert` and `retract`, leanT^AP relies on Prolog's clause indexing scheme and backtracking mechanism; it exploits the power of Prolog's inference engine as much as possible, without changing the database.

Although (or better: because) this prover is so small, it shows striking performance: For example, nearly all of Pelletier's problems [5] can be solved (the only exceptions are Problem No. 34 “Andrew's Challenge” and No. 47 “Schubert's Steamroller”). Running on a SUN SPARC 10 workstation they are proven in less than 0.2*sec*, most of them in less than 0.01*sec*. Some of the theorems, like Problem 38, are quite hard: the (complex) tableau-based prover $_3T^AP$ [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, see [7, 6].

This shows that a first-order calculus based on free-variable semantic tableaux can be efficiently implemented in Prolog with minimal means. Moreover, leanT^AP can be further improved without breaking the rules of lean theorem proving [2], e.g., by taking into account “universal variables”.



One could regard programs like Satchmo and leanT^AP as a Prolog hack. However, we think they demonstrate more than tricky use of Prolog: they show why the philosophy of “lean theorem proving” is interesting: 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.

It is interesting to consider the principle of lean deduction w.r.t. applications. Deduction systems like ours have their limits, in that many problems are solvable with complex and sophisticated theorem provers but not with an approach like leanT^AP. However, when applying deduction in practice, this might not be relevant at all: the above figure oversimplifies but shows the point; the x-axis gives a virtual value of the complexity of a problem, and the y-axis shows the runtime required for finding a solution. The two graphs give the performance of lean and of large deduction systems. We are better off with a system like leanT^AP below a certain degree of problem complexity: it is compact, easier adaptable to an application, and also faster because it has less overhead than a huge system. Between a break-even point, where sophisticated systems become faster, and the point where small systems fail, it is not immediately clear which approach to favor: adaptability can still be a good argument for lean deduction. For really hard problems, a sophisticated deduction system is the only choice. This last area, however, could indeed be neglectable, depending on the requirements of an application: if few time can be allowed, we cannot treat hard problems by deduction at all. Thus, lean deduction can be superior in all cases—depending on the concrete application.

References

1. B. Beckert, S. Gerberding, R. Hähnle, and W. Kernig. The tableau-based theorem prover ${}_3T^AP$ for multiple-valued logics. In *Proc., CADE-11, Albany/NY*, LNCS. Springer, 1992.
2. B. Beckert and J. Posegga. leanT^AP: Lean tableau-based theorem proving. Submitted, 1993.
3. M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 1990.
4. R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In *Proc., CADE-9, Argonne*, LNCS. Springer, 1988.
5. F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *JAR*, 2:191–216, 1986.
6. J. Posegga. Compiling proof search in semantic tableaux. In *Proc., ISMIS-7, Trondheim, Norway*, LNCS. Springer, 1993.
7. M. E. Stickel. A prolog technology theorem prover. In *Proc., CADE-9, Argonne*, LNCS. Springer, 1988.

Appendix: The Theorem Prover leanT^AP

We will briefly describe the tableau-based theorem prover leanT^AP listed at the beginning and explain its behavior. We assume familiarity with free-variable tableaux (e.g. [3]) and the basics of programming in Prolog.

For the sake of simplicity, we restrict our considerations 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. However, Skolemization has to be implemented carefully to achieve the highest possible performance.[1]

We 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 $p(0) \wedge (\forall n(\neg p(n) \vee p(s(n))))$).

A single Prolog predicate `prove(Fml,UnExp,Lits,FreeV,VLim)` implements our prover; it 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 (Prolog variables, which might be bound to a term). A positive integer `VLim` is used to initiate backtracking; it is an upper bound for the length of `FreeV`.

The prover is started with the goal `prove(Fml,[],[],[],VLim)`, which succeeds if `Fml` can be proven inconsistent without using more than `VLim` free variables on each branch.

If a conjunction "A and B" is to be expanded, then "A" is considered first and "B" is put in the list of not yet expanded formulæ (Clause 1). For disjunctions we split the current branch and prove two new goals (Clause 2).

Handling universally quantified formulæ (γ-formulæ) requires a little more effort (Clause 3). We first check the number of free variables on the branch. Backtracking is initiated if the depth bound `VLim` 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` (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), 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`.

Clause 4 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 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. Note, that the implication "`->`" after binding `Neg` introduces an implicit cut: this prevents generating double negation when backtracking.

Clause 5 is reached if Clause 4 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.

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 third clause enumerates closing substitutions during backtracking.

[1] A Prolog program for computing an optimized negation normal form, as well as leanT^AP's source code, is available upon request from the authors.