Logic Programming as a Basis for Lean Deduction: Achieving 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

1 Introduction

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 and that this does not suit the needs of many applications.[1] It is hard to see how to apply these systems – 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 deduction* 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. Logic programming languages provide an ideal tool for implementing lean deduction, as they offer a level of abstraction that is close to the needs for building first-order deduction systems.

The Prolog program shown in Figure 1, called lean$T^{A}P$ (Beckert & Posegga, 1994b), is an instance of such a lean deduction system: it implements a complete and sound theorem prover for first-order logic in Skolemized negation normal form. The underlying calculus is based on free-variable semantic tableaux (Fitting, 1990) (we shall explain the program in Section 2).

Our approach surely does not lead to deduction systems that are superior to highly sophisticated theorem provers like Otter (McCune, 1990) or Setheo (Letz *et al.*, 1992); these are better on solving difficult problems. However, many applications do not require deduction which is as complex as the state of the art in automated theorem proving. Furthermore, there are often strong constraints on the time allowed for deduction. In such areas our approach can be extremely useful: it offers high inference rates on simple to moderately complex problems and a high degree of adaptability.

Another important argument for lean deduction is safety: It is easily possible to verify the couple of lines of standard Prolog implementing lean$T^{A}P$ (Beckert & Posegga, 1994a; Posegga & Schmitt, 1995); verifying thousands of lines of C code, however, is hard—if not impossible—in practice.

2 lean$T^{A}P$: The Program

We will briefly describe the basic working principle of the tableau-based theorem prover lean$T^{A}P$ shown in Figure 1.[2] We assume familiarity with free-variable tableaux (e.g. (Fitting, 1990)) 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 strong 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 (Beckert *et al.*, 1993).[3]

*Proc. Workshop on Logic Programming, University of Zürich, Switzerland. Oct. 1994

[1]Empirical evidence for this claim can easily be given: Hardware and Software Verification are usually listed as the most important application areas with practical relevance of Automated Deduction. However, when looking closer, one realizes that the implemented systems in this area either use interactive provers, or application-specific developments. The classical, stand-alone theorem provers seem to be not flexible enough to be integrated into such systems.

[2]The interested reader is might wish to consult (Beckert & Posegga, 1994a) for a more detailed explanation.

[3]A Prolog program for computing an optimized negation normal form, as well as lean$T^{A}P$'s source code, is available upon request from the authors.

```
1 prove((A,B),UnExp,Lits,FreeV,VLim) :- !,
       prove(A,[B|UnExp],Lits,FreeV,VLim).
2 prove((A;B),UnExp,Lits,FreeV,VLim) :- !,
       prove(A,UnExp,Lits,FreeV,VLim),
       prove(B,UnExp,Lits,FreeV,VLim).
3 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).
4 prove(Lit,_,[L|Lits],_,_) :-
      (Lit = -Neg; -Lit = Neg) ->
      (unify(Neg,L); prove(Lit,[],Lits,_,_)).
5 prove(Lit,[Next|UnExp],Lits,FreeV,VLim) :-
      prove(Next,UnExp,[Lit|Lits],FreeV,VLim).
```

Figure 1: leanT^AP: The Basic Version of the Program

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 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.

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`.

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.

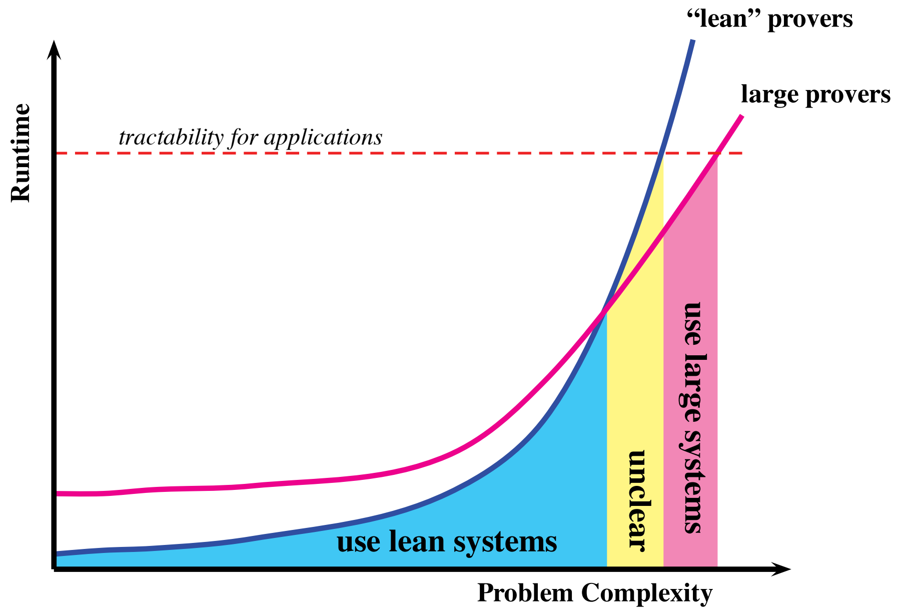

Figure 2: Lean versus Large Deduction Systems

Although (or better: because) this prover is so small, it shows striking performance: For example, nearly all of Pelletier's problems (Pelletier, 1986) can be solved (the only exceptions are Problem No. 34 "Andrews' Challenge" and No. 47 "Schubert's Steamroller"). Running on a SUN SPARC 10 workstation they are proven in less than $0.2sec$, most of them in less than $0.01sec$. Some of the theorems, like Problem 38, are quite hard: the (complex) tableau-based prover ${}_3T^AP$ (Beckert *et al.*, 1992), 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 (Stickel, 1988; Posegga, 1993a; Posegga, 1993b).

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 (Beckert & Posegga, 1994a), e.g., by taking into account "universal formulæ".

3 The Idea Behind Lean Deduction

Satchmo (Manthey & Bry, 1988) can be seen as the earliest application of what we call lean deduction. lean$T^AP$was written in the same spirit, but does not have much in common with Satchmo, besides that it is also a small Prolog program.[4]

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:

[4]Both programs differ significantly from a logical, as well as from an implementation-oriented point of view: leanT^AP is based on free-variable semantic tableaux and works for general first-order formulæ, whereas Satchmo applies a model elimination-like calculus to range-restricted formulæ in clausal form (CNF). Satchmo extensively uses `assert` and `retract` for implementing first-order deduction, whereas leanT^AP relies on Prolog's clause indexing scheme and backtracking mechanism, and does not change Prolog's database at all.

Figure 2 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.

4 Conclusion & Outlook

We showed the principle of lean deduction and presented leanT^AP, an instance of it implemented in standard Prolog. It was argued that lean deduction can ease integrating deduction into applications, as it offers a new basis for implementations: Rather than going for highly-sophisticated, high performance theorem provers, a flexible and easily adaptable approach is used. For this, calculi based on semantic tableaux seem to be better suited than resolution-based systems.

Lean deduction lies somewhere between Logic Programming and Theorem Proving: it is programming logics, rather than logic programming. Both Automated Deduction and Logic Programming can obviously contribute a lot to it: experience and know-how in implementing calculi can be gained from Automated Deduction, and Logic Programming can provide the appropriate implementation basis. We used only standard Prolog, but it is easy to see that we could take advantage of many enhancements to Prolog. It will be subject to future research to explore this to a greater extend.

References

BECKERT, B., & POSEGGA, J. 1994a. leanT^AP: Lean Tableau-Based Deduction. *Journal of Automated Reasoning*. (to appear).

BECKERT, B., & POSEGGA, J. 1994b. leanT^AP: lean, tableau-based theorem proving. *In: Proc. CADE-12*. LNCS. Nancy, France: Springer.

BECKERT, B., GERBERDING, S., HÄHNLE, R., & KERNIG, W. 1992. The Tableau-Based Theorem Prover ${}_3T^AP$ for Multiple-Valued Logics. *In: Proc. CADE-11*. LNCS. Albany, NY: Springer.

BECKERT, B., HÄHNLE, R., & SCHMITT, P. H. 1993. The Even More Liberalized δ-Rule in Free Variable Semantic Tableaux. *In: Proc., 3rd Kurt Gödel Colloquium (KGC)*. LNCS. Brno, Czech Republic: Springer.

FITTING, M. 1990. *First-Order Logic and Automated Theorem Proving*. Springer.

LETZ, R., SCHUMANN, J., BAYERL, S., & BIBEL, W. 1992. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, **8**(2), 183–212.

MANTHEY, R., & BRY, F. 1988. SATCHMO: A Theorem Prover Implemented in Prolog. *In: Proc. CADE-9*. LNCS. Argonne, ILL: Springer.

MCCUNE, W. W. 1990. *Otter 2.0 Users Guide*. Tech. rept. ANL–90/9. Argonne National Laboratories, Mathematics and Computer Science Division.

PELLETIER, F. J. 1986. Seventy-Five Problems for Testing Automatic Theorem Provers. *Journal of Automated Reasoning*, **2**, 191–216.

POSEGGA, J. 1993a. Compiling the Proof Search in Semantic Tableaux. *In: 7th ISMIS*. LNCS. Trondheim, Norway: Springer.

POSEGGA, J. 1993b. *Deduktion mit Shannongraphen für Prädikatenlogik erster Stufe*. St. Augustin, Germany: infix-Verlag.

POSEGGA, J., & SCHMITT, P. H. 1995. Implementing Tableau-based Deduction. *In: Handbook of Tableau-based Methods in Automated Deduction*. Oxford University Press. (to appear).

STICKEL, M. E. 1988. A Prolog Technology Theorem Prover. *In: Proc. CADE-9*. LNCS. Argonne, ILL: Springer.