LOGIC PROGRAMMING AS A BASIS FOR LEAN AUTOMATED DEDUCTION *

BERNHARD BECKERT AND JOACHIM POSEGGA

▷ The idea of *lean deduction* is to achieve maximal efficiency from minimal means. Every possible effort is made to eliminate overhead. 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. In this paper we describe the principle of lean deduction and present leanTAP, an instance of it implemented in standard Prolog. ◁

1. INTRODUCTION

The idea underlying *lean automated 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. The result is a clearly structured implementation, that is easy to modify, easy to adapt to concrete needs of an application, and easy to integrate with other components of a system.

The design of lean deduction theorem provers is motivated by the observation, that—compared to the amount of research carried out—Automated Deduction is

*This paper is based on a talk that was given at the *Workshop on Logic Programming*, University of Zürich, Switzerland, October 1994. Its focus is on the general idea of lean deduction and the relation to Logic Programming. The reader who is more interested in the program leanTAP, that is presented as an example, is referred to [2]. leanTAP's source code, as well as a Prolog program for computing an optimized negation normal form, is available in the *WWW* at `http://i12www.ira.uka.de/leantap`.

Address correspondence to Bernhard Beckert, Institute for Logic, Complexity, and Deduction Systems, University of Karlsruhe, D-76128 Karlsruhe, Germany. E-mail: `beckert@ira.uka.de`; and Joachim Posegga, Deutsche Telekom AG, Research Centre, FZ122h, D-64276 Darmstadt, Germany, E-mail: `posegga@fz.telekom.de`.

THE JOURNAL OF LOGIC PROGRAMMING

©Elsevier Science Publishing Co., Inc., 1993
655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

little applied in practice. Most researchers working in automated reasoning believe that there are many useful applications of the techniques developed, but few attempts are made to actually build these applications.

We see one reason for this in that implementation-oriented research in Automated Deduction favors huge and highly complex systems, which does not suit the needs of many applications: most existing automated theorem provers must be seen as black boxes from a user's point of view; the algorithms implemented for carrying out deductions are usually highly sophisticated, and the interaction of the various parameters influencing the search for proofs is often far from being obvious. Furthermore, the low-level interfaces of these systems are often poorly developed, which makes it very hard to integrate such a tool into a given software environment, and even harder to adapt the implementation to meet concrete needs. From the software engineer's point of view, current automated reasoning tools mostly come as monolithical systems.[1]

When looking at the field of software engineering, however, one realizes that monolithical systems become less and less important: the interest is focussed since quite a while on software components [9], and interoperability (see e.g. [10, 8] for standards) of open, distributed systems. One idea behind both is to provide smaller pieces of software that interact with each other, rather than using a monolithical approach to building software. The main driving force behind these approaches is to increase the adaptability of software, and to reduce the complexity of the individual software components for gaining more reliable and more flexible systems. The underlying motivation is very similar to what let us start our work on leanTAP.

2. leanTAP: AN INSTANCE OF LEAN DEDUCTION

The Prolog program shown in Figure 1 is an instance of a lean deduction system: it implements a complete and sound theorem prover for first-order formulae in skolemized negation normal form. The underlying calculus is based on free-variable semantic tableaux (see for example [3]).

The program is a variant of leanTAP [2], where a few changes have been made to increase readability. We shall now briefly explain the program, but refer the reader for a detailed discussion to [2].

leanTAP exploits the power of Prolog's inference engine as much as possible, using its 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, we restrict our considerations to first-order formulae 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 [1].

We use Prolog syntax for first-order formulae: 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,

[1] There are of course also other approaches like the KEIM system [4], an extension of Common Lisp that provides an extensible toolbox for the development of deduction systems.

```
1 refute((A,B),UnExp,Lits,FreeV,VLim) :- !,
     refute(A,[B|UnExp],Lits,FreeV,VLim).
2 refute((A;B),UnExp,Lits,FreeV,VLim) :- !,
     refute(A,UnExp,Lits,FreeV,VLim),
     refute(B,UnExp,Lits,FreeV,VLim).
3 refute(all(X,Fml),UnExp,Lits,FreeV,VLim) :- !,
    \+ length(FreeV,VLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[all(X,Fml)],UnExp1),
    refute(Fml1,UnExp1,Lits,[X1|FreeV],VLim).
4 refute(Lit,_,Lits,_,_) :- closed(Lit,Lits).
5 refute(Lit,[Next|UnExp],Lits,FreeV,VLim) :-
    refute(Next,UnExp,[Lit|Lits],FreeV,VLim).
6 closed(-L,[L1|T]) :- !, (unify(L,L1) ; closed(-L,T)).
7 closed(L,[-L1|T]) :- unify(L,L1) ; closed(L,T).
```

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

a first-order formula is represented by a Prolog term (for example, the formula $p(0) \wedge (\forall n)(\neg p(n) \vee p(s(n)))$ is represented by `(p(0),all(N,(-p(N);p(s(N)))))`).

The Prolog predicate `refute(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 `refute(Fml,[],[],[],VLim)`, which succeeds if `Fml` can be proven inconsistent without using more than `VLim` free variables on each tableau branch.[2]

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 formulae 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). The 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 formulae (Clause 1). For disjunctions we split the current branch and prove two new goals (Clause 2).

Handling universally quantified formulae (γ-formulae) 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

---

[2] If one wants to avoid committing on the number `VLim`, the predicate `refute` can be called with iterative deepening on `VLim`. The standard solution in Prolog for this is:

```
inc_refute(Fml,VarLim) :- refute(Fml,[],[],[],VarLim).
inc_refute(Fml,VarLim) :- NewVarLim is VarLim + 1,
                          inc_refute(Fml,NewVarLim).
```

When started with `inc_prove(Fml,N)`, the prover searches with the values `N`, `N+1`, ... for `VarLim`.

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 clause of `refute` which is not determinate. It succeeds if the current formula `Lit` (which must be a literal) is contradictory to one of the other literals `Lits` on the current branch. In this case, the current branch is closed. The test is implemented by the predicate `closed` (see below).

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.

Clauses 6 and 7 implement the test for a closed branch used in Clause 4; it is basically a variant of the standard `member`-predicate with sound unification.

leanTAP has two choice points: one is selecting between the last two clauses of `refute`, which means closing a branch or extending it. The second choice point within each of the two clauses of `closed` enumerates closing substitutions during backtracking.

3. ADVANTAGES OF LEAN DEDUCTION

Although leanTAP does show surprising performance[3] it certainly does not outperform highly sophisticated theorem provers like Otter [7] or Setheo [5]. 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.

The latter is the actual strength of our approach: leanTAP is a very simple and clear implementation; similar to Satchmo [6], it offers an alternate view on Automated Deduction: rather than being confronted with a highly complex, monolithical system, one can use an open implementation, that is easy to understand and easy to adapt to and embed into applications.[4] Since leanTAP is based on semantic tableaux, it can, furthermore, be adapted to other logics, for example modal or temporal logics.

Another important argument for lean deduction is safety: It is easily possible to verify the couple of lines of standard Prolog implementing leanTAP [2]; verifying thousands of lines of C code, however, is hard—if not impossible—in practice.

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

[3] For example, nearly all of Pelletier's problems [11] can be solved. 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*.

[4] Whilst Satchmo and leanTAP are quite related from this meta level point of view, they differ significantly from a logical point of view: leanTAP implements a standard, free-variable semantic tableaux calculus for formulae in negation normal form, whereas Satchmo uses a model generation-like calculus for range-restricted formulae in clausal form. Both calculi show quite different behavior w.r.t. the proof search.

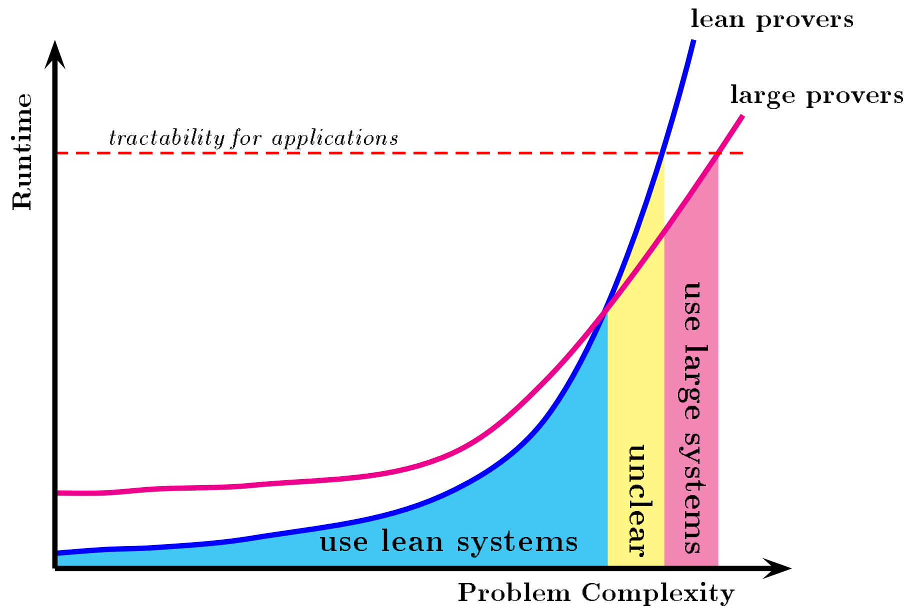

Figure 1. Lean versus Large Deduction Systems

with complex and sophisticated theorem provers but not with an approach like leanTAP. However, when applying deduction in practice, this might not be relevant at all: Figure 1 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 leanTAP 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. LOGIC PROGRAMMING: THE BASIS FOR LEAN DEDUCTION

The basis for lean deduction is an appropriate implementation language, that offers a level of abstraction that is close to the logic one wishes to implement. Logic Programming languages offer the ideal basis, since they come equipped with mechanisms for representing terms, variables, substitutions, etc. This is one important prerequisite for leanTAP-like implementations.

A second, equally important issue is that the search strategy used for deduction is supported: standard Prolog, as used for leanTAP implements depth-first search, thus there is no need to re-code this for controlling the proof search. This is the most important point for the efficiency of leanTAP: it relies on the efficient implementation of backtracking in the underlying Prolog system.

Lean deduction lies somewhere between Logic Programming and Theorem Proving: it is programming logics, rather than logic programming. Experience and know-how in implementing calculi is taken from Automated Deduction, and Logic Programming provides the ideal 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.

We would like to thank Deepak Kapur for valuable comments on an earlier version of this paper.

REFERENCES

1. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. The even more liberalized δ-rule in free variable semantic tableaux. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Proceedings, 3rd Kurt Gödel Colloquium (KGC), Brno, Czech Republic*, LNCS 713, pages 108–119. Springer, 1993.
2. Bernhard Beckert and Joachim Posegga. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.
3. Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 1990.
4. Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg Siekmann. KEIM: A toolkit for automated deduction. In A. Bundy, editor, *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France*, LNCS 814, pages 807–810. Springer, 1994.
5. Reinhold Letz, Johann Schumann, Stephan Bayerl, and Wolfgang Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.
6. Rainer Manthey and François Bry. SATCHMO: A theorem prover implemented in Prolog. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction (CADE)*, LNCS, pages 415–434, Argonne, Ill, 1988. Springer.
7. William W. McCune. *OTTER Users' Guide, Version 2.0.* Argonne National Laboratory, 1990.
8. Microsoft. *Microsoft OLE 2 Design Team: Object Linking & Embedding. OLE 2.01 Design Specification.* Microsoft Inc., 1993.
9. Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, 1992.
10. Object Management Group. *The Common Object Request Broker: Architecture and Specification, Rev. 2.0.* OMG (Object Management Group), Framingham, MA, 1995.
11. Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.