Journal of Automated Reasoning **15:** 339–358, 1995.

leanTAP: Lean Tableau-based Deduction

BERNHARD BECKERT and JOACHIM POSEGGA
Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, 76128 Karlsruhe, Germany
e-mail: {beckert,posegga}@ira.uka.de, WWW: http://i12www.ira.uka.de/

(Received: 20 July 1994)

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.

Key words: logic for artificial intelligence, theorem proving, automated deduction, semantic tableaux, first-order logic, Prolog.

1. Introduction

The Prolog program listed in the abstract implements a complete and sound theorem prover for first-order logic; it is based on free-variable semantic tableaux (Fitting, 1990). We call this *lean deduction*: the idea is to achieve maximal efficiency from minimal means. We will see that the program is indeed very efficient – not *although* but *because* it is extremely short and compact.

Our approach surely does not lead to a deduction system that is superior to highly sophisticated systems like Otter (McCune, 1990) or Setheo (Letz et al., 1992); these are better on solving difficult problems. However, many applications do not require deduction that is as complex as the state of the art in automated theorem proving can handle. Furthermore, there are often strong constraints on the time allowed for deduction. Our approach can be particularly useful in such areas: 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 a couple of lines of standard Prolog; verifying thousands of lines of C code, however, is hard – if not impossible – in practice.

Satchmo (Manthey and Bry, 1988) 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 formulae in clausal form (CNF). Range-restrictedness can be avoided with some extra effort, but the restriction to clausal form is crucial to Satchmo's underlying calculus. Many problems become much harder when translating them to clausal form, so it seems better to avoid CNF and to preserve position and scope of quantifiers.* 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.

PAPER OUTLINE

The paper is organized as follows: Section 2 discusses our implementation and explains the underlying idea. In Section 3 we present a simple but efficient method for computing an optimized negation normal form (NNF). After giving some performance data in Section 4, we describe in Sections 5 and 6 how the program can be improved by using the "universal formula" mechanism; in Section 7 we sketch the proof for leanTAP's soundness and completeness.

We draw conclusions from our research and give an outlook to future applications of lean theorem proving in Section 8. Finally, in an appendix, we briefly survey the history of tableau-based theorem provers.

Through the paper we assume familiarity with free-variable tableaux for classical first-order logic (see, e.g., Fitting, 1990) and the basics of programming in Prolog (O'Keefe, 1990). All source code given in this paper is available via the *World Wide Web* on

```
http://i12www.ira.uka.de/~posegga/leantap/leantap.html.
```

2. The Program

The idea behind leanTAP is to exploit the power of Prolog's inference engine as much as possible. We rely on Prolog's clause indexing scheme and backtracking mechanism, and 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 closed first-order formulae 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 (cf. Section 3). We will use Prolog syntax for first-order formulae: atoms are Prolog terms, "-" is negation, ";" disjunction, and "," conjunction.

* Using a definitional CNF (Eder, 1992) helps at most partially: it avoids exponential growth of formulae for the price of introducing some redundancy into the proof search. Extending the scope of quantifiers to clause level, however, cannot be avoided.

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`. This is the case if the formula is inconsistent. 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 (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**) "A and B" is to be expanded, then "A" is considered first and "B" is put in the list of not yet expanded formulae:

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

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

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

Handling universally quantified formulae (γ-formulae) 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`‡, 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`:

* Formally, we should say "the formula that is represented by the Prolog term bound to `Fml`". However, we will simply write "the formula `Fml`" in the sequel.

** Due to R. Smullyan, conjunctive type formulae are called α-formulae in the semantic tableaux framework.

‡ 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.

```
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).
```

Recall that "\+" denotes Prolog's negation as failure. `copy_term(+Term,-Copy)` makes a copy of `Term` by replacing each distinct variable in `Term` by a new variable that occurs nowhere else in the system, and unifies `Copy` with the result.

The next clause of `prove` closes branches; it is the only one that is not determinate. Note that it will be entered only 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.

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

Note, that the implication "->" after binding `Neg` introduces an implicit cut: 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.

The enumeration of closing substitutions is controlled with the limit `VarLim`: if the limit is reached before a closed tableau has been found, the clause for universally quantified formulae fails and leanT^AP searches for alternate substitutions for closing branches. However, leanT^AP will never change a given value for `VarLim`, thus the program is only complete in the sense that it will find a proof if one with less than `VarLim` γ-rule applications on each branch exists.

It is important in practice that the limit is not chosen too high, as the search space grows exponentially with `VarLim`. A good solution for this problem is to simply wrap the call to the predicate `prove` in Prolog code that implements iterative deepening. The standard solution in Prolog for this is:

* In contrary to Prolog's built-in unification "=", the predicate `unify` implements sound unification, i.e., unification with occur check. Most Prolog systems provide `unify` as a library predicate.

```
inc_prove(Fml,VarLim) :- prove(Fml,[],[],[],VarLim).

inc_prove(Fml,VarLim) :- NewVarLim is VarLim + 1,
                         inc_prove(Fml,NewVarLim).
```

The prover is then started with `inc_prove(Fml,N)` and searches with the values `N`, `N` $+1$, ... for `VarLim`.

3. Computing a Negation Normal Form

The prover above works only for formulae in negation normal form (NNF), since we did not implement all tableau rules for general formulae. If we want to use leanTAP for formulae that are not in NNF, we can either add clauses for `prove` that implement the corresponding rules, or apply the conversion into NNF as a preprocessing step. The first alternative means merging the derivation of NNF into the proof search, the latter separating it from the proof search. In both cases, the same operations are carried out and we do not gain anything from carrying them out simultaneously. On the other hand, it is reasonable to keep the proof search as simple as possible; we will therefore separate both issues and derive an NNF in advance.

Most operations for deriving NNF are straightforward. What is not straightforward is coming up with a good skolemization. This is one reason we give a complete Prolog implementation of the conversion. The second is that we show how to optimize the NNF without extra cost by changing the sequence of disjunctively connected formulae.

Recall that the conversion into NNF is linear w.r.t. the length of a formula not containing equivalences. With equivalences, it can be implemented with quadratic effort (Eder, 1992), but, for the sake of simplicity, we will use a naive version which is in the worst case exponential for equivalences.

The predicate used for computing a negation normal form is

```
nnf(+Fml,+FreeV,-NNF,-Paths)
```

`Fml` is the formula to be transformed, `FreeV` is the list of free variables occurring in `Fml`, `NNF` is bound to the Prolog term representing the computed NNF of `Fml`, and `Paths` is bound to the number of disjunctive paths in `NNF` (resp. `Fml`). We will see soon what this latter information is good for.

The goal for computing the NNF of `Fml` is `nnf(Fml,[],NNF,_)`. We implement a more convenient syntax for first-order formulae, using as logical connectives "v" (disjunction), "&" (conjunction), "=>" (implication), and "<=>" (equivalence).

The first clause of the predicate `nnf` corresponds to the standard rules in semantic tableaux; nothing exciting is done – we just use tautologies for rewriting formulae:

```
nnf(Fml,FreeV,NNF,Paths) :-
```

```
(Fml = -(-A)       -> Fml1 = A;
 Fml = -all(X,F)   -> Fml1 = ex(X,-F);
 Fml = -ex(X,F)    -> Fml1 = all(X,-F);
 Fml = -(A v B)    -> Fml1 = -A & -B;
 Fml = -(A & B)    -> Fml1 = -A v -B;
 Fml = (A => B)    -> Fml1 = -A v B;
 Fml = -(A => B)   -> Fml1 = A & -B;
 Fml = (A <=> B)   -> Fml1 = (A & B) v (-A & -B);
 Fml = -(A <=> B) -> Fml1 = (A & -B) v (-A & B)),!,
nnf(Fml1,FreeV,NNF,Paths).
```

For universally quantified formulae, we add the quantified variable to `FreeV` and compute the NNF of the scope:

```
nnf(all(X,F),FreeV,all(X,NNF),Paths) :- !,
    nnf(F,[X|FreeV],NNF,Paths).
```

Skolemization has to be carried out very carefully, since straightforward skolemizing can easily hinder finding a proof. Fitting (1990) proposes to insert a Skolem-term containing all variables that appear free on a branch. This is correct, but too restrictive: it often delays closing of inconsistent branches. The current state of the art (Beckert et al., 1993) is less restrictive: It suffices to use a Skolem-term that is unique (up to variable renaming) to the existentially quantified formula. This term only holds the free variables occurring in the formula (and not all free variables on the current branch as Fitting proposes).* An ideal candidate for such a term is the formula itself.** This can be implemented in Prolog in the following way:

```
nnf(ex(X,Fml),FreeV,NNF,Paths) :- !,
        copy_term((X,Fml,FreeV),(Fml,Fml1,FreeV)),
        copy_term((X,Fml1,FreeV),(ex,Fml2,FreeV)),
        nnf(Fml2,FreeV,NNF,Paths).
```

We generate a copy `Fml1` of the scope `Fml`; none of the free variables in `Fml1` are renamed (they occur in `FreeV`), except the existentially quantified variable `X`; `Fml` is inserted for it. `Fml` contains all the free variables that have to occur in a Skolem-term and we use it for this purpose. We do not need to create a new function symbol for skolemization, since we will assume that disjoint sets of predicate and function symbols are used. The second `copy_term` goal substitutes the existentially quantified variable that is free in `Fml1` by the (arbi-

* From a logical perspective, our version of skolemization results in a stronger calculus than the one proposed by Fitting (1990); both calculi are complete, but the shortest proofs in Fitting's calculus are sometimes longer.

** We thank Christian Fermüller for pointing out, that this is closely related to the ϵ-formulae described in (Hilbert and Bernays, 1939, §1).

trary) constant ex, since we do not want to introduce new free variables when skolemizing.*

The next clause is routine, besides counting disjunctive paths:

```
nnf(A & B,FreeV,(NNF1,NNF2),Paths) :- !,
    nnf(A,FreeV,NNF1,Paths1),
    nnf(B,FreeV,NNF2,Paths2),
    Paths is Paths1 * Paths2.
```

The number of disjunctive paths in a formula (i.e., the number of branches a fully expanded tableau for it will have) is used when handling disjunctions: we put the less branching formula to the left. That way the number of choice points during the proof search is reduced, since leanTAP will expand the left part of a disjunction first.

```
nnf(A v B,FreeV,NNF,Paths) :- !,
    nnf(A,FreeV,NNF1,Paths1),
    nnf(B,FreeV,NNF2,Paths2),
    Paths is Paths1 + Paths2,
    (Paths1 > Paths2 -> NNF = (NNF2;NNF1);
                        NNF = (NNF1;NNF2)).
```

The last clause will match literals:

```
nnf(Lit,_,Lit,1).
```

4. Performance

Although (or better: because) the prover is so small, it shows striking performance. Table I shows experimental results for a subset of Pelletier's problems (Pelletier, 1986). We placed the negated theorem in front of the axioms and used the above program for computing negation normal form.

Some of the theorems, like Problem 38, are quite hard: the 3TAP prover (Beckert et al., 1992), for instance, needs more than ten times as long. If leanTAP 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 (Stickel, 1988; Posegga, 1993a; Posegga, 1993b).

Schubert's Steamroller (Pelletier No. 47) cannot be solved; this is no surprise, since the problem is designed for forward chaining based on clauses. It can be proven in tableau-based systems only if good heuristics for selecting γ-formulae are used. Using a queue, as in our case, is not sufficient. We console ourselves with Problem No. 38, which is barely solvable in a comparable time by CNF-based provers.

* Since this might be a bit hard to understand for people not used to programming in Prolog, we give an example: the Prolog query `nnf(ex(X,p(X,Y)),[Y],NNF,_)` will succeed and bind `NNF` to `p(p(ex,Y),Y)` (`Fml` is bound to `ex(X,p(X,Y))`, `Fml1` to `p(p(X,Y),Y)`, and `Fml2` to `p(p(ex,Y),Y)`).

TABLE I. leanTAP's performance for Pelletier's problems.

No.	Limit	Branches		Time	No.	Limit	Branches		Time
	VarLim	closed	tried	msec		VarLim	closed	tried	msec
17	1	14	14	0	32	3	10	10	10
18	2	1	1	9	33	1	11	11	0
19	2	4	6	0	34	5	267	792	700
20	6	3	3	9	35	4	1	1	0
21	2	8	8	0	36	6	3	3	0
22	2	7	14	9	37	7	8	8	9
23	1	4	4	0	38	4	90	101	210
24	6	33	33	9	39	1	2	2	0
25	3	5	5	0	40	3	4	5	0
26	3	16	17	0	41	3	4	5	0
27	4	8	8	0	42	3	5	5	9
28	3	5	5	0	43	5	18	18	109
29	2	11	11	9	44	3	5	5	10
30	2	4	4	9	45	5	17	17	39
31	3	5	5	0	46	5	53	63	59

The runtime has been measured on a SUN SPARC 10 workstation with SICStus Prolog 2.1; "0 msec" means "not measurable". The times reflect the search for a proof with iterative deepening on `VarLim` and include NNF derivation.

Pelletier No. 34 (also called "Andrews' Challenge") is the hardest problem; it can only be solved if exactly the required value of 5 for `VarLim` is chosen. 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 15 minutes.

5. Universal Variables in Formulae

One of the major problems with implementing a first-order tableau calculus is to control the application of universal quantifiers or γ-formulae. These generate the free variables in a tableau, which may be instantiated for closing branches. In tableaux, these free variables are not implicitly universally quantified as it is for instance the case with variables in clauses when using a resolution calculus. Free variables in tableaux are *rigid*: the same substitution must be applied to all occurrences of the variable in the whole tableau.

From a more formal point of view, this situation is closely related to what is usually called the *strong consequence relation.*

DEFINITION 1 (Strong consequence relation). Let ϕ, ψ be first-order formulae;

$$\phi \models\!\!\!\circ\ \psi$$

if for all interpretations I and for all variable assignments β:

$$\text{if } val_{I,\beta}(\phi) = true \text{ then } val_{I,\beta}(\psi) = true.$$

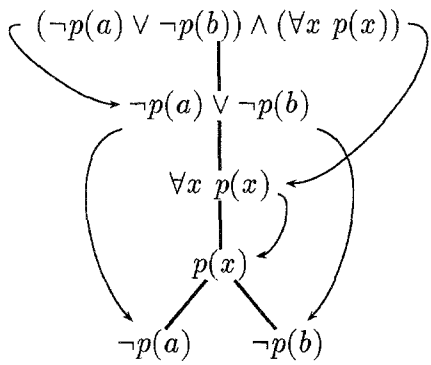

Fig. 1. An example for universal formulae.

(Note, that for example $p(x) \not\models^{\circ} (\forall x p(x))$, but $p(x) \models (\forall x p(x))$, where "$\models$" denotes the weak consequence relation.)

Suppose we have a branch B with a formula $\phi(x)$ on it in a tableau. Assume further that the expansion of the tableau then proceeds with creating new branches; some of these branches contain occurrences of x. For closing the generated branches, we must use the same substitution for x on all of them. Figure 1 gives an example for the situation: the tableau cannot be closed immediately, since no single substitution closes both branches simultaneously. To find a proof, we have to apply the γ-rule again and create another instance of $p(x)$.

For particular branches it could be the case that $B \models^{\circ} (\forall x\, \phi(x))$. This is for instance the case in Figure 1. On these branches, we can use different substitutions for x. The tableau in Figure 1 then closes immediately. Recognizing such situations and exploiting them yields shorter tableau proofs, since fewer rule applications for closing a tableau are needed.

This is the idea behind the heuristic discussed in this section. We present a method that detects universal variables in formulae in some cases.*

DEFINITION 2 (Universal formula). Suppose ϕ is a formula on some tableau branch B. *ϕ is universal on B with respect to the variable x* if

$$B \models^{\circ} (\forall x\, \phi) \ .^{**}$$

Now, we can use a new rule for closing branches that takes this definition into account.

* See (Beckert and Hähnle, 1992) for details; we only give a slightly simplified account here.

** In the sequel, we will often refer to a formula ϕ that is universal on a branch B w.r.t. a variable x just by "the universal formula ϕ", and to the variable x by "the universal variable x" (if the context is clear).

DEFINITION 3 (Closed tableau). A tableau consisting of k branches B_i $(1 \leqslant i \leqslant k)$ is closed if there are

1. a substitution σ,
2. literals l_i, $\bar{l}_i \in B_i$, and
3. substitutions σ_i, such that
   (a) $l_i\sigma_i$ and $\bar{l}_i\sigma_i$ are complementary, and
   (b) if $\sigma_i(x) \neq \sigma(x)$ then both l_i and $\bar{l}_i$ are universal on B_i w.r.t. x.

With this definition of closed tableaux it is possible that a tableau is closed after less applications of expansion rules than in the standard free-variable tableau calculus. Thus, the calculus is strengthened.

The problem of recognizing universal formulae is of course undecidable in general. However, a wide and important class can be recognized quite easily: assume there is a sequence of tableau rule applications that does not contain a disjunctive rule (i.e., the tableau does not branch). All formulae that are generated by this sequence are universal w.r.t. the free variables introduced by the sequence. Substitutions for these variables can be ignored, since the corresponding inference steps could be repeated arbitrarily often to generate new instances of the universal variables (without generating new branches).

More formally, we use the following lemma.

LEMMA 1. *A formula ϕ on a branch B is universal w.r.t. x if ϕ was put on B by either*

1. *applying a γ-rule and x is the free variable introduced by the application of this rule, or*
2. *applications of non-branching rules to a formula $\psi \in B$, where ψ is universal on B w.r.t. x.*

A proof is immediate, since the criteria of Definition 2 are implied.

Recognizing the above subset of universal formulae in the leanTAP program, can be implemented by keeping a list of (in this sense) universal variables for each formula. This information is used to rename the universal variables occurring in literals, such that their instantiation does not affect the rest of the tableau. This renaming "simulates" universal quantification of the variable that is renamed; it is carried out when expanding a disjunction.

For this, the arity of `prove` is extended from 5 to 7:

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

The use of all parameters but `UnivV` and `DisV` remains unchanged. `UnivV` is a list of the universal variables in `Fml`. `DisV` is a Prolog term containing all variables on the current branch that are *not* universal in one of the formulae (we will call these "disjunctive variables"). Each unexpanded formula in `UnExp` will have the list of its universal variables attached. The Prolog functor ":" is used for this.

The prover is now started with the goal

```
prove(Fml,[],[],[],[],[],VarLim)
```

to prove the inconsistency of `Fml`. We will discuss the extended program by explaining the differences to our previous version.

All universal variables of a conjunction are universal for each of its components:

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

Disjunction destroys universality: the universal variables of a disjunction are not universal to its components. The tableau is split and the universal variables become non-universal on both resulting branches. We therefore add them to `DisV` by creating a new Prolog term.* Universal variables occurring in the literals on the branch are renamed by `copy_term`. Renaming allows to instantiate the universal variables differently on the two resulting branches.

```
prove((A;B),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
    copy_term((Lits,DisV),(Lits1,DisV)),
    prove(A,UnExp,Lits,(DisV+UnivV),FreeV,[],VarLim),
    prove(B,UnExp,Lits1,(DisV+UnivV),FreeV,[],VarLim).
```

When introducing a new variable by the quantifier rule, this variable becomes universal for the scope (it may lose that status if a disjunction in the scope is expanded; see above).

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

The next clause remains unchanged except for having two more parameters.

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

Recall that the sixth parameter of `prove` holds the universal variables of the current formula (not of the whole branch). Thus, when extending branches we must change this argument:

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

* We could use a list, but creating a new term by "+" (an arbitrary functor) is faster.

TABLE II. leanTAP's performance with the universal formula mechanism.

No.	Limit	Branches		Time	No.	Limit	Branches		Time
	VarLim	closed	tried	msec		VarLim	closed	tried	msec
17	1	14	14	10	32	3	10	10	10
18	2	1	1	0	33	1	11	11	10
19	2	3	3	10	34	5	79	79	109
20	6	3	3	9	35	2	1	1	0
21	2	8	8	0	36	6	3	3	0
22	2	4	4	0	37	7	8	8	30
23	1	4	4	0	38	4	90	101	489
24	6	33	33	39	39	1	2	2	0
25	3	5	5	0	40	3	4	5	0
26	3	16	17	19	41	3	4	5	9
27	4	8	8	10	42	3	5	5	9
28	3	5	5	10	43	5	18	18	179
29	2	11	11	19	44	3	5	5	19
30	2	4	4	0	45	5	17	17	79
31	3	5	5	10	46	5	53	63	189

The runtime has been measured on a SUN SPARC 10 workstation with SICStus Prolog 2.1; "0 msec" means "not measurable".

6. Performance with the Universal Formula Mechanism

Problem No. 34 can now be solved faster, as Table II shows. The runtime for other problems (like 38) increased, since some overhead is involved with maintaining universal variables. Note that Problem No. 22 works better now: we need only four instead of seven branches.

7. Proving leanTAP's Soundness and Completeness

7.1. PRELIMINARIES

One of the advantages of leanTAP's compactness is that it is possible to formally prove its correctness, i.e., its soundness and completeness. Nevertheless, due to space restrictions, we will not give a detailed proof here, but only a proof sketch. We present the theorems that have to be proven and the main arguments that may be used.

The proof makes use of the well-known fact that free variable semantic tableaux are a sound and complete calculus. In addition, we assume (and do not prove) the Prolog compiler (resp. interpreter) to be correct, as well as the implementation of library predicates.

First, we formally define the free variable tableau calculus, using a slightly non-standard representation:* Tableaux are multi-sets of multi-sets of first-order formulae; as usual, the branches of a tableau are implicitly disjunctively connected, and the formulae on a branch are implicitly conjunctively connected.

DEFINITION 4. A tableau is a (finite) multi-set of tableau branches, where a tableau branch is a (finite) multi-set of first order formulae.

Three types of rules can be applied to a tableau to derive a new one: expansion rules, the closure rule, and the substitution rule. The expansion rules are the classical α-, β- and γ-rules for formulae in NNF (only using our set notation). The closure rule removes closed branches instead of just marking them as being closed.

DEFINITION 5. Let T be a tableau, $B \in T$ a branch of T, and $\phi \in B$ a formula on B.

Expansion rules: The tableau may be derived from T that is constructed by removing the branch B from T and replacing it by one (resp. two) new branch:

$$\begin{array}{lll} (B \setminus \{\phi\}) \cup \{\psi_1, \psi_2\} & \text{if } \phi = \psi_1 \wedge \psi_2, & (\alpha) \\ (B \setminus \{\phi\}) \cup \{\psi_1\} \quad \text{and } (B \setminus \{\phi\}) \cup \{\psi_2\} & \text{if } \phi = \psi_1 \vee \psi_2, & (\beta) \\ B \cup \{\psi(y)\} \quad (\text{where } y \text{ is a new variable}) & \text{if } \phi = (\forall x)\psi(x). & (\gamma) \end{array}$$

Closure rule: If B is closed (i.e., if there are complementary literals $l, \bar{l} \in B$), then $T \setminus \{B\}$ may be derived from T.

Substitution rule: The tableau $T\sigma$ may be derived from T, where σ is any substitution that does not instantiate bound variables in T (including the empty substitution).

During leanTAP's proof search, the current `prove` goal together with the `prove` goals on the Prolog goal stack** represent the tableau that has been computed so far. A goal `prove(Fml,UnExp,Lits,FreeV,VarLim)` represents the tableau branch consisting of the formulae in `Fml`, `UnExp` and `Lits`, that still has to be closed.

7.2. SOUNDNESS

The soundness theorem to be proven is as follows.

* We stress that this calculus differs from classical free variable tableaux (e.g., Fitting, 1990) only in notation and the way tableaux are represented.

** The state a Prolog computation has reached is usually represented as a list (stack) $[G_1, \ldots, G_k]$ of atomic formulae (called goals), and a substitution σ of the Prolog variables occurring in this list. σ is the answer substitution computed up to that point. For our purposes, however, it is not necessary to separate the substitution from the goals; we therefore consider $[G_1\sigma, \ldots, G_k\sigma]$ to be the current goal stack.

THEOREM 1. *If* `Fml` *is bound to a closed first-order formula* ϕ *in NNF, and the goal* `prove(Fml,[],[],[],VarLim)` *succeeds, then* ϕ *is inconsistent.*

The proof is based on the soundness of free variable tableaux.

FACT 2. If ϕ is a closed first-order formula ϕ in NNF, and the empty tableau (the empty set) can be derived from the initial tableau $\{\{\phi\}\}$ by applying a finite sequence of the rules from Definition 5, then ϕ is inconsistent.

Given Fact 2, it suffices to validate the following statements to prove Theorem 1:

- If `Fml` is bound to ϕ, the initial goal `prove(Fml,[],[],[],VarLim)` represents the initial tableau $\{\{\phi\}\}$.
- Whenever leanT^AP changes the set of `prove` goals on the Prolog goal stack (i.e., derives a new tableau), this corresponds to an application of one of the tableau rules from Definition 5.*
- leanT^AP terminates successfully only when the goal stack is empty, that is, when the empty tableau has been derived.

7.3. COMPLETENESS

The completeness theorem to be proven is as follows.

THEOREM 3. *If* ϕ *is an inconsistent first-order formula in NNF, then there is an* $n \geqslant 0$ *such that, if* `Fml` *is bound to* ϕ *and* `VarLim` *is bound to* n*, then the goal* `prove(Fml,[],[],[],VarLim)` *succeeds.*

Central to the proof of Theorem 3 is the notion of *fully expanded tableaux* and that of a sequence of tableau derivations that ends with a fully expanded tableau.

DEFINITION 6. A sequence $T_0, \ldots, T_n$ of tableaux is a fully expanding tableau sequence w.r.t. the limits p and q ($p, q \geqslant 0$) if T_{i+1} has been derived from T_i by using one of the rules from Definition 5 ($1 \leqslant i \leqslant n$), and:

1. only expansion rules have been applied in the sequence,
2. there are only literals and γ-formulae in T_n,
3. if there is a γ-formula ϕ on a branch $B \in T_i$ ($0 \leqslant i \leqslant n$) that is one of the first q formulae that have been added to B (or were initially present on B), then the γ-rule has been applied at least p times to this occurrence of ϕ.

α- and β-formulae are removed from a tableau once their according rule has been applied to them. Therefore, Condition 2 in Definition 6 is equivalent to:

* leanT^AP's last clause does not change the tableau, but only its internal representation.

There is no α- or β-formula in the tableau that the according rule has not been applied to.

Using the notion of fully expanding sequence, one can formulate the well-known completeness theorem for free variable semantic tableaux in the following way:

FACT 4. If the first-order formula ϕ that is in NNF is inconsistent, then there are limits p and q $(p, q \geqslant 0)$ such that for any sequence $T_0, \ldots, T_n$ of tableaux that begins with the initial tableau $T_0 = \{\{\phi\}\}$ and that is a fully expanding sequence w.r.t. p and q, there is a substitution σ such that each branch of the tableau $T_n\sigma$ is closed.

That is, there are (i) literals l_i and $\bar{l}_i$ on each branch B^i $(1 \leqslant i \leqslant m)$ of T_n, and (ii) substitutions μ_i that are more general than σ such that $l_i\mu_i$ and $\bar{l}\mu_i$ are complementary.

The tableau sequences computed by leanTAP are not fully expanding, because leanTAP closes branches immediately that contain complementary literals. However, we can achieve this with a variant leanTAP′ of the program that is identical to leanTAP except that (i) the fourth clause, which closes branches, is omitted; and (ii) there is an additional clause

```
prove(Fml,UnExp,Lits,_,_) :-
    write(['The branch consisting of ', Fml, UnExp, Lits,
           ' is part of the fully expanded tableau']).
```

at the end of leanTAP′. This last clause is needed, because we do not want the construction of the expansion sequence to fail when a branch is fully expanded.

Now, if `Fml` is bound to the inconsistent formula ϕ in NNF, `VarLim` is chosen high enough, and leanTAP′ is started by the goal `prove(Fml,[],[],[],VarLim)`, then it constructs a fully expanding sequence $T'_0, \ldots, T'_{n'}$ w.r.t. arbitrary limits $p, q \geqslant 0$, in particular w.r.t. the limits that exist according to Fact 4. Therefore, the tableau $T'_{n'}$ is closed (as described in Fact 4) if only `VarLim` is high enough. The proof of this can be based on the following arguments (that have to be validated):

- leanTAP′ neither applies substitutions nor closes and removes branches (Definition 6, Cond. 1),
- the α- and the β-rule are applied as often as possible (Definition 6, Cond. 2),
- the list `UnExp` implements a priority queue. Therefore, the γ-rule is applied arbitrarily often to each γ-formula, if only `VarLim` is high enough (Definition 6, Cond. 3),
- the computation of tableau branches and tableaux terminates, since with each step either the formulae on the branch become less complex, the length of `FreeV` increases, or the number of formulae in `UnExp` decreases.

It remains to be proven that the original leanTAP constructs a closed tableau as well and, in addition, actually closes the branches. To do this, we change – in two steps – the fully expanding sequence $T'_0, \ldots, T'_{n'}$ constructed by leanTAP', such that the resulting sequence is constructed by the original leanTAP and ends with the empty tableau:

First, all expansions of branches are removed from the sequence that already contain the pair $l_i, \bar{l}_i$ of closing literals (since leanTAP does not expand such branches). It is easy to check that the last tableau in the resulting sequence is closed in the same way as $T'_{n'}$, using the same literals and substitutions.

In a second step, substitution and closure rule applications are inserted into the sequence. As soon as closing literals $l_i, \bar{l}_i$ occur on a branch $B \in T'_i$, the substitution μ_i is applied to T'_i, and the closed branch $B\mu_i$ is removed by using the closure rule.

Obviously, the resulting tableau sequence $T_0, \ldots, T_n$ $(n \leqslant n')$ ends with the empty tableau $T_n = \emptyset$. By induction on i one proves that after a finite number of the original leanTAP's computation steps (and possibly after backtracking, if there are choice points), the `prove` goals exactly represent the tableau T_i. For $i = n$ this immediately implies that leanTAP derives the empty tableau, that is, terminates with success.*

For the induction proof, one has to validate that applying closing substitutions and deleting closed branches does not affect the expansion of the rest of the tableau, i.e., of those branches that have not been closed yet. The order in which formulae are chosen for expansion remains the same.

8. Conclusion and Outlook

We showed how a first-order calculus based on free-variable semantic tableaux can be efficiently implemented in Prolog with minimal means. The proposed implementation is surprisingly efficient, especially if universal formulae are taken into account.

One could regard leanTAP as a Prolog hack. However, we think it demonstrates more than tricky use of Prolog: it shows that semantic tableaux can be efficiently implemented with little effort. Among other benefits, this feature makes leanTAP ideal for classroom use.

Furthermore, 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 may be other possibilities to construct an empty (closed) tableau; in that case it is not obvious which one leanTAP will find first.

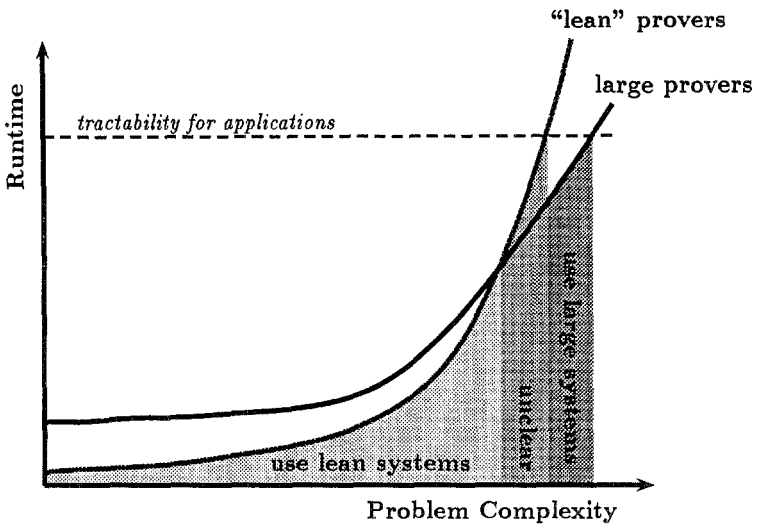

Fig. 2. Lean vs. large deduction systems.

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 leanTAP. However, when applying deduction in practice, this might not be relevant 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 run time 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: leanTAP is compact, easier to adapt 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 at least 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 little 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*.

* Researchers in automated reasoning often regret/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; systems such as leanTAP do give a good starting point here.

There is still room for improvement without sacrificing simplicity and/or elegance of our approach. We can, for instance, use an additional preprocessing step that translates a negation normal form into a graphical representation of a fully expanded tableau (see (Posegga, 1993a) for details). This can be implemented equivalently simply and requires only linear effort at run time. The prover itself then becomes smaller, since no compound formulae are present any more and all branches are already fully developed. The speedup will not be dramatic, but it will be considerable. Furthermore, we can implement the compilation principle described by Posegga (1993a): 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.

Appendix

A Brief Historical Survey of Tableau-based Provers

Compared with resolution, few attempts have been made in the past to implement tableau-based calculi; thus we can take the risk of presenting a brief survey (which, nevertheless, is likely to be incomplete).*

The first tableau-based theorem prover was developed in the late fifties by Dag Prawitz, Håkan Prawitz, and Neri Voghera (Prawitz et al., 1960). It ran on a computer named Facit EDB (manufactured by AB Åvidabergs Industrier). The tableau calculus implemented was already quite similar to today's versions; it did not, however, use free variables. This prover was perhaps the earliest for first-order logic at all.**

At about the same time, Hao Wang implemented a prover for first-order logic that was based on a sequent calculus similar to semantic tableaux (Wang, 1960). The program ran on IBM 704 computers.

Ewa Orłowska implemented a calculus that can be seen as tableau-based in 1967 on a GIER digital computer‡. The calculus was based on deriving *if-then-else* normal forms rather than disjunctive normal forms. Only the propositional part of the calculus was implemented.

We are not aware of any implementation-oriented research around tableaux in the seventies; there have been a number of theoretic contributions to tableau calculi but nothing seems to have been implemented.

In the eighties, the research lab of IBM in Heidelberg, Germany, was a major driving force of tableau-based deduction. Wolfgang Schönfeld developed a prover

* It is restricted to approaches for formulae of first-order logic in non-clausal form.

** Actually, Prawitz et al. implemented a calculus for first-order logic without function symbols; that, however, has the same expressiveness as full first-order logic.

‡ The GIER (Geodaetisk Instituts Elektroniske Regnemaskine) was produced by Regnecentralen in Copenhagen (Denmark) in the early sixties.

within a project on legal reasoning (Schönfeld, 1985). It was based on free-variable semantic tableaux and used unification for closing branches. A few years later Peter Schmitt developed the THOT theorem prover at IBM (Schmitt, 1987); this was also an implementation of free-variable tableaux and part of a project aiming at natural language understanding. Both implementations have been carried out in Prolog. Based on experiences with the THOT theorem prover, the development of the 3TAP system started around 1990 at Karlsruhe University (Beckert et al., 1992); the project was funded by IBM Germany and carried out by Peter Schmitt and Reiner Hähnle. The 3TAP prover was again written in Prolog and implemented a calculus for free-variable tableaux, both for classical first-order logic with equality as well as for multi-valued logics. This program can be seen as the direct ancestor of leanTAP.

Besides the line of research outlined above there was also other work on tableau-based deduction in the eighties: Oppacher and Suen published their well-known paper on the HARP theorem prover in 1988 (Oppacher and Suen, 1988). This prover was implemented in LISP and is probably the best-known instance of a tableau-based deduction system. Another implementation, the Helsinki Logic Machine (HLM), is a Prolog program that actually implements about 60 different calculi, among them semantic tableaux for classical first-order logic, nonmonotonic logic, dynamic logic, and autoepistemic logic. Approximately at the same time a tableau-based prover was implemented at Karlsruhe University by Thomas Käufl (Käufl and Zabel, 1990); the system, called "Tatzelwurm," implemented classical first-order logic with equality but did not use a calculus based on free variables. Its main purpose was to be used as an inference engine in a program verification system.

Since 1990, the interest in tableau-based deduction continuously increased, and we will not try to continue our survey beyond this date. From 1992 onwards, the activities of the international tableau community are quite well documented, as annual workshops were started; we refer the interested reader to the workshop proceedings (Fronhöfer et al., 1992; Basin et al., 1993; Broda et al., 1994).*

References

Basin, D., Fronhöfer, B., Hähnle, R., Posegga, J., and Schwind, C.: 2nd Workshop on Theorem Proving with Analytic Tableaux and Related Methods. Technical Report 213, Max-Planck-Institut für Informatik, Saarbrücken, Germany, May 1993.

Beckert, B. and Hähnle, R.: An improved method for adding equality to free variable semantic tableaux, in D. Kapur (ed.), *11th Int. Conf. on Automated Deduction (CADE)*, Lecture Notes in Computer Science, Springer-Verlag, Albany, NY, 1992, pp. 507–521.

Beckert, B., Gerberding, S., Hähnle, R., and Kernig, W.: The tableau-based theorem prover 3TAP for multiple-valued logics, in *11th Int. Conf. on Automated Deduction (CADE)*, Lecture Notes in Computer Science, Springer-Verlag, Albany, NY, 1992, pp. 758–760.

* Proceedings of subsequent workshops will be published within Springer's LNCS series.

Beckert, B., Hähnle, R., and Schmitt, P. H.: The even more liberalized δ-rule in free variable semantic tableaux, in G. Gottlob, A. Leitsch, and D. Mundici (eds), *3rd Kurt Gödel Colloquium (KGC)*, Lecture Notes in Computer Science, Springer-Verlag, Brno, Czech Republic, 1993, pp. 108–119.

Broda, K., D'Agostino, M., Goré, R., Johnson, R., and Reeves, S.: 3rd Workshop on Theorem Proving with Analytic Tableaux and Related Methods, Technical Report TR-94/5, Imperial College London, Department of Computing, London, England, April 1994.

Eder, E.: *Relative Complexities of First-Order Calculi*. Artificial Intelligence. Vieweg-Verlag, 1992.

Fitting, M. C.: *First-Order Logic and Automated Theorem Proving*, Springer-Verlag, 1990.

Fronhöfer, B., Hähnle, R., and Käufl, T.: Workshop on Theorem Proving with Analytic Tableaux and Related Methods. Technical Report 8/92, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany, March 1992.

Hilbert, D. and Bernays, P.: *Grundlagen der Mathematik II*, Vol. 50 of Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete, Springer-Verlag, 1939.

Käufl, T. and Zabel, N.: Cooperation of decision procedures in a tableau-based theorem prover, *Revue d'Intelligence Artificielle* **4**(3) (1990).

Letz, R., Schumann, J., Bayerl, S., and Bibel, W.: SETHEO: A high-performance theorem prover, *J. Automated Reasoning* **8**(2) (1992), 183–212.

Manthey, R. and Bry, F.: SATCHMO: A theorem prover implemented in Prolog, in E. Lusk and R. Overbeek (eds), *9th Int. Conf. on Automated Deduction (CADE)*, Lecture Notes in Computer Science, Springer-Verlag, Argonne, IL, 1988, pp. 415–434.

McCune, W. W.: Otter 2.0 Users Guide, Technical Report ANL–90/9, Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL, March 1990.

O'Keefe, R. A.: *The Craft of Prolog*, MIT Press, 1990.

Oppacher, F. and Suen, E.: HARP: A tableau-based theorem prover, *J. Automated Reasoning* **4** (1988), 69–100.

Pelletier, F. J.: Seventy-five problems for testing automatic theorem provers, *J. Automated Reasoning* **2** (1986), 191–216.

Posegga, J.: Compiling proof search in semantic tableaux, in *7th Int. Symp. on Methodologies for Intelligent Systems (ISMIS)*, Lecture Notes in Computer Science, Springer-Verlag, Trondheim, Norway, 1993, pp. 67–77.

Posegga, J.: *Deduktion mit Shannongraphen für Prädikatenlogik erster Stufe*, Infix-Verlag, St. Augustin, Germany, 1993.

Prawitz, D., Prawitz, H., and Voghera, N.: A mechanical proof procedure and its realization in an electronic computer, *ACM* **7**(1–2) (1960), 102–128.

Schmitt, P. H.: The THOT Theorem Prover, Technical Report 87.9.7, IBM Germany, Scientific Center, Heidelberg, Germany, 1987.

Schönfeld, W.: Prolog extensions based on tableau calculus, in *9th Int. Joint Conf. on Artificial Intelligence, Los Angeles*, Vol. 2, 1985, pp. 730–733.

Stickel, M. E.: A Prolog technology theorem prover, in E. Lusk and R. Overbeek (eds), *9th Int. Conf. on Automated Deduction (CADE)*, Lecture Notes in Computer Science, Springer-Verlag, Argonne, IL, 1988, pp. 752–753.

Wang, H.: Toward mechanical mathematics, *IBM J. Research and Development* **4**(1) (1960).