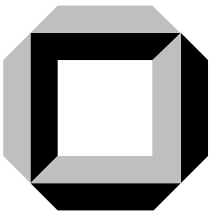
Implementing Semantic Tableaux

Joachim Posegga
Peter H. Schmitt

Interner Bericht 12/96



Universität Karlsruhe
Fakultät für Informatik

Implementing Semantic Tableaux

Joachim Posegga & Peter H. Schmitt

Contents

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
76128 Karlsruhe, Germany
WWW: `http://emmy.ira.uka.de/`
Email: `posegga@fz.telekom.de`, `pschmitt@ira.uka.de`

This report will appear as a chapter in the
Handbook of Tableau-based Methods in Automated Deduction,
D. Gabbay, M. D'Agostino, R. Hähnle, and J. Posegga (eds.),
with KLUWER ACADEMIC PUBLISHERS.
Electronic availability will be discontinued
after final acceptance for publication is obtained.

What is it people want to hear about an implementation? Most likely they will be content to hear that it works, or better that it works extremely well. Who cares on the sun deck of a cruise ship what happens in the engine-room? There is some excuse for this attitude: implementation is often associated with nitty-gritty details, with cumbersome work-arounds caused by insufficencies of the programming language, or with genial short cuts through several levels of abstraction in the specification. This is definitely not what we want to present in this chapter. It is tempting to join the group in their deck chairs and talk elegantly about how one would theoretically realize an implementation, step-by-step refining the top level specification and carefully weighing-up all design decisions. This is a possible approach. For this time we decided on a presentation half-way between the two portrayed alternatives. We will present runnable code, but in an easily accessible language that also has the advantage that some of the important procedures used in theorem proving algorithms are already available as built-ins or library functions. We are speaking of Prolog. The reader is invited to type the theorem proving programs he will find in this chapter into his favorite Prolog system and enjoy playing around with them.

The programs are based on, or inspired by, the the leanTAP theorem prover [BP94]. The idea behind leanTAP is to implement logical calculi by minimal means. This has two advantages: Firstly, the resulting programs are small, which makes it easier to understand them. Second, they provide an ideal starting point as they can be easily modified or adapted to specific needs. Furthermore, they are more than mere toy systems and surprisingly fast.

We will provide extensive comments on these programs and in one case also a complete soundness and correctness proof. Different alternatives for representing the tableaux and for organizing the proof search will be considered and exemplified by small Prolog programs.

A draw back of this approach is that the reader will be required to understand Prolog. But let us hasten to assure that acquaintance with the basic ideas of Prolog will suffice, all of which may be found e.g. on the first 22 pages of [CM81]. In addition we will need in the soundness and correctness proof a formal semantics of the underlying programming language. To this end we will review below the basic computation model of Prolog, the computation tree. This offers another possibility for the reader to acquire an understanding for this language or to consolidate it.

The plan for this chapter is as follows: Section 1 fixes a couple of assumptions we will make for discussing our approaches to implementing tableaux. This involves some Prolog-oriented issues, as well as certain points about the tableaux calculi underlying our implementations. In Section 2 an algorithm for deriving Skolemized negation normal form is presented. The

input formulæ for the programs given subsequently will be in Skolemized negation normal form. Section 3 presents the first and simplest version of our theorem prover. The program has been proposed in [BP94]; here we recall it and prove its soundness and completeness. Section 4 extends the program by a powerful heuristic called *universal formulæ*. Building upon a theorem prover that represents tableaux as graphs in Section 5, we present a compilation-based approach to tableau-based deduction in Section 6. Section 7 discusses lemmata in tableau calculi which leads us to Binary Decision Diagrams (BDDs). Section 8 functions as a conclusion by giving some ideas on how one can build upon the presented programs when working towards his or her own implementation.

1 Preliminaries

There are a couple of issues one needs to consider when carrying out implementations of deduction systems. Clearly, the concrete calculus that is to be implemented and the language chosen for an implementation are of most importance.

We have chosen to use Prolog as the implementation language used in this chapter. The reason for this is pragmatic: Prolog is a very convenient language for implementing first-order reasoning, since the primitives of Prolog are already quite close to first-order logic. This allows one to program in a very elegant and short style, as we will see in the sequel. Nevertheless there are some subtle points to be considered if we want to obtain efficient code:

As we want to achieve efficient code, we will want, to take advantage of the strengths of Prolog systems. One is that Prologs depth-first search with backtracking is usually implemented very efficiently. Fortunately, this is also a well-suited search strategy for implementing deduction. Unfortunately, Prolog's search strategy is incomplete, since it chooses whatever comes first in the database instead of having a *fair* selection scheme. Since we want to implement logically *complete* deduction systems, we will have to overcome this drawback; one way to tackle it is switching to a bounded depth-first search. The desired completeness can be obtained by successively increasing a depth bound.[1]

It is also important to observe that Prolog's efficiency is strongly enhanced by indexing on the first argument position of the clause head. Thus, putting the right information in the first argument pays off.

The Prolog code we will give in the sequel is standard Prolog (in Edinburgh syntax) and should run on most Prolog systems[2]. We assume that

[1] The other choice would have been to implement fairness. But given the facts that no convincing fairness criteria are known and the difficulty in changing Prolog's search strategy without losing efficiency, this is not a viable alternative.

[2] The code was developed and tested with Sicstus Prolog , but runs without changes

the following Prolog predicates, user defined or otherwise, are available:

`append/3. append(L1,L2,L3)` succeeds if `L3` is the result of appending the lists `L1` and `L2`.

`unify/2. unify(T1,T2)` unifies the Prolog terms `T1` and `T2` by *sound* unification.

Logical formulæ will be represented by Prolog terms as follows:

Prolog atom	atomic formula
`-`	negation
`;`	disjunction
`,`	conjunction
`all(X,F)`	universal quantification with `X` a Prolog variable and `F` the scope of quantification.

Thus `(p(a),all(X,(-p(X);p(f(X)))))` stands for

$$p(a) \wedge \forall x(\neg p(x) \vee p(f(x))).$$

Furthermore, we assume that no variable is used twice for quantification within a set of input formulæ, e.g. a formula of the form

$$\forall x(q(x) \rightarrow q(x)) \wedge \forall x(p(x))$$

should be avoided. This assumption is in fact not neccessary for using the programs we will present, but it makes them more consise and easier to unterstand.

As we now have presented a rough idea of the means by which we will implement deductions, let us say a word on the calculi we will implement.

We will consider theorem provers for classical first-order logic without equality and will use a tableau calculus in its free-variable version Usually tableau calculi are set up for general formulæ with many logical connectives. We decided from a presentational perspective to use only formulæ in negation normal form using only universal quantifiers.[3] Arbitrary formulæ will be reduced to this format in a preprocessing step.

The issue whether preprocessing or normalization by tableau rules during tableau expansion is the better choice is not resolved at the moment, and maybe there will never be a definite answer. But separating preprocessing from the actual proof search certainly leads to a much clearer presentation. It is of course possible to extend the implementations we will give below to non-negation normal form formulæ without preprocessing, if one wishes to do so.

---

with Quintus Prolog and Eclispe; other Prolog dialects might require little changes to our programs.

[3] Note, that we do not require formulæ to be in prenex normal form.

2 Preprocessing

2.1 Computing a Negation Normal Form

Recall that the conversion into negation normal form is linear w.r.t. the length of a formula not containing equivalences[4]. Most operations for deriving negation normal form 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 negation normal form without extra cost by changing the order of disjunctively connected formulæ.

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 negation normal form 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.

We implement a convenient syntax for first-order formulæ, using as logical connectives "`v`" (disjunction), "`&`" (conjunction), "`=>`" (implication), and "`<=>`" (equivalence).

The Prolog query we are going to use for computing the negation normal form of a closed formula bound to `Fml` is `nnf(Fml,[],NNF,_)` [5] . The corresponding program is given in Figure 1. The first clause of the predicate `nnf` (lines 1–11) corresponds to the standard rules in semantic tableaux; nothing exciting is done—we just use tautologies for rewriting formulæ. For universally quantified formulæ, we add the quantified variable to `FreeV` to compute the negation normal form of the scope (12–13).

Skolemization has to be carried out very carefully, since straightforwardly Skolemizing can easily hinder finding a proof: In the first edition of [Fit90] Skolem-termsw containing all variables that appear free on a branch are inserted; this is correct, but too restrictive: it often prevents inconsistent branches from closing. The current state of the art [BHS93] 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 needs to hold the free variables occurring in the formula. An ideal candidate for such a term is the formula itself. This way of Skolemization has actually been known for more than fifty years: it resembles the ϵ-formulæ described in [HB39, §1]. Lines 14–16 show how this can be elegantly implemented in Prolog.

---

[4] If the formula contains equivalences, its negation normal form becomes exponential when computed in a naive way; more clever algorithms result in an at most quadratic NNF [Ede92].

[5] The symbol _, called anonymous variable, is a convenient way to name Prolog variables, you don't care about

```
% Rewriting logical connectives:
1  nnf(Fml,FreeV,NNF,Paths) :-
2     (Fml = -(-A)      -> Fml1 = A;
3      Fml = -all(X,F)  -> Fml1 = ex(X,-F);
4      Fml = -ex(X,F)   -> Fml1 = all(X,-F);
5      Fml = -(A v B)   -> Fml1 = -A & -B;
6      Fml = -(A & B)   -> Fml1 = -A v -B;
7      Fml = (A => B)   -> Fml1 = -A v B;
8      Fml = -(A => B)  -> Fml1 = A & -B;
9      Fml = (A <=> B)  -> Fml1 = (A & B) v (-A & -B);
10     Fml = -(A <=> B) -> Fml1 = (A & -B) v (-A & B)),!,
11     nnf(Fml1,FreeV,NNF,Paths).
% Universal Quantification:
12 nnf(all(X,F),FreeV,all(X,NNF),Paths) :- !,
13    nnf(F,[X|FreeV],NNF,Paths).
% Skolemization:
14 nnf(ex(X,Fml),FreeV,NNF,Paths) :- !,
15    copy_term((X,Fml,FreeV),(Fml,Fml1,FreeV)),
16    nnf(Fml1,FreeV,NNF,Paths).
% Conjunctions:
17 nnf(A & B,FreeV,(NNF1,NNF2),Paths) :- !,
18    nnf(A,FreeV,NNF1,Paths1),
19    nnf(B,FreeV,NNF2,Paths2),
20    Paths is Paths1 * Paths2.
% Disjunctions:
21 nnf(A v B,FreeV,NNF,Paths) :- !,
22    nnf(A,FreeV,NNF1,Paths1),
23    nnf(B,FreeV,NNF2,Paths2),
24    Paths is Paths1 + Paths2,
25    (Paths1 > Paths2 -> NNF = (NNF2;NNF1);
26                        NNF = (NNF1;NNF2)).
% Literals:
27 nnf(Lit,_,Lit,1).
```

Fig. 1. Computing Negation Normal Form

The `copy_term` goal in line 15 is all that is needed: its result is that the existentially quantified variable `X` is replaced by the original scope `Fml`. Note, that we cannot implement this with `X=Fml`: it would result in a cyclic term.

In Sicstus Prolog `copy_term` behaves as if it were defined by

```
copy_term(X,Y) :-
       assert('copy of'(X)),
       retract('copy of'(Y)).
```

If `t` is a prolog term with the Prolog variables `X,A,B,C` then the query `copy_term((X,t,[A,B,C]),(X1,Y,[A,B,C]))` succeeds binding `Y` to a copy of `t` where `X` is replaced by `X1`. The variable `X1` is new, in the sense that any subsequent binding of `X` does not affect `X1`. For example, the query

```
copy_term(f(X),f(Y)),X=a,Y=b.
```

succeeds.

The free variables `FreeV` must appear as an argument in both parameters in line 15, since we do *not* want to rename them.

From a logical point of view, this might look a bit odd, as we turn predicate symbols into function symbols when Skolemizing in this way. However, it works under the assumption that disjoint sets of predicate and function symbols are used. This is usually the case; if not, we can simply "wrap" the inserted scope in a new function symbol.

The next clause (17–20) is routine, besides counting disjunctive paths. 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 (12–26): we put the less branching formula to the left. That way the number of choice points during the proof search is reduced, since leanTAP expands the left formula first.

The last clause will match literals and is again straightforward.

3 A Simple and Efficient Tableaux-based Theorem Prover

In this section we present a simple Prolog implementation of the tableau method for formulæ in Skolemized negation normal form. This is not only a pedagogical device, it really works: type the code from Figure 2 — we will refer to this program by the symbol leanTAP in the following — into your favorite Prolog system and it will run extremely well at least on small examples. A second objective of this section is the proof that leanTAP is a correct implementation. More precisely, we will eventually show

Theorem 3.0.1.

1. *The* leanTAP *program terminates on all inputs.*
2. *If the query* `prove(`*fml*`,`*h*`,[],[],`*d*`)` *to the program* leanTAP *returns success as an answer, where fml is a formula, h is a list of formulæ and d is a natural number, than* $\neg$*fml is a logical consequence of h.*
3. *If* $\neg$*fml is a logical consequence of h then there is a natural number d such that the query* `prove(`*fml*`,`*h*`,[],[],`*d*`)` *to* leanTAP *terminates with success.*

```
% Conjunction:
prove((A,B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,[B|UnExp],Lits,FreeV,VarLim).
% Disjunction:
prove((A;B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,UnExp,Lits,FreeV,VarLim),
    prove(B,UnExp,Lits,FreeV,VarLim).
% Universal Quantification:
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).
% Closing Branches:
prove(Lit,_,[L|Lits],_,_) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L); prove(Lit,[],Lits,_,_)).
% Extending Branches:
prove(Lit,[Next|UnExp],Lits,FreeV,VarLim) :-
    prove(Next,UnExp,[Lit|Lits],FreeV,VarLim).
```

Fig. 2. A Complete and Sound Tableau Prover

In the proof of this theorem we will not deal with the logical consequence relation directly but make use of the completeness theorem established in a previous chapter:

$$\neg fml \text{ is a logical consequence of } h$$
$$\text{iff}$$
$$\text{there is a closed tableau for } [fml \mid h].$$

A tableau T for a list L of formulæ starts with a non ramifying branch B, the nodes in B being labelled by the formulæ in L in some order.

Before going into details we will briefly outline the working principle of the `leanTAP` program:

`prove(Fml,[],[],[],VLim)` 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. A positive integer `VarLim` is used to initiate backtracking; it is an upper bound for the length of `FreeV`.

We will number clauses of the leanTAP-program by the line in which they start in the listing of Figure 2. Clause 1 handles conjunctions: the first conjunct is selected, the other is put in the list of not yet expanded formulæ. Handling disjunctions, clause 3 splits the current branch and two new goals have to be proven.

Universally quantified formulæ require a little more effort. Clause 6 uses the built-in predicate `length(X,Y)`, which succeeds if `X` is a list of length `Y`. The symbol `\+` denotes negation in Sicstus Prolog. The built-in predicate `copy_term` has already been explained above.

Application of clause 6 initiates backtracking if the depth bound `VarLim` is reached. Otherwise, we generate a "fresh" instance of the formula with `copy_term`, without renaming the free variables in `FreeV`. The original γ-formula is stored for subsequent use, and the renamed scope becomes the current formula.

Clause 11 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. Clause 11 incorporates the design decision to look for complementary formulas only at the level of literals. This suffices for completeness, but closure with arbitrary complementary formulas can be faster.[6]

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

3.1 Proving Completeness & Correctness of leanTAP

To reason about the execution of the program leanTAP we need an operational semantics of Prolog as a programming language. We will use for this purpose the computation tree, T_P, associated with a Prolog program P. This concept will be explained in detail below. Computation trees are a very simple model for an operational semantics of Prolog and other descriptions are available, see e.g. [BR94]. But leanTAP uses the cut "`!`" only in obvious ways, negation only in line 7 and none of the meta programming features at all. Only the "`->`" construct in line 12 may not be completely standard. Thus, the overhead for a deeper operational model does not pay off.

To begin our explanation of the concept of the computation tree, T_P, we remark that the nodes of this tree are labelled by the states of computation that arise during the execution of a Prolog program P. At this level of

[6] As an example, take $\Phi \wedge \neg\Phi$, where Φ is an arbitrarily complex formula. In practice, however, such phenomena occur very rarely.

abstraction a state of computation consists of a list, *goallist*, of atomic formulæ, called goals, and a substitution σ of the Prolog variables occurring in this list. If a goal list $[F_1, \ldots, F_k]$ and a substitution σ are attached to a node $\mathbf{n}$, then it is really the list of formulæ $[\sigma(F_1), \ldots, \sigma(F_k)]$ that we consider. But it will prove convenient to separate this information into a formula part and a substitution part.

Now we turn to the question, what are the successor nodes of a node $\mathbf{n}$ labelled with a pair $(goallist, \sigma)$? Among the list of goals still to be solved Prolog always chooses the first one. Putting aside for the moment the case of the empty list of goals we write $goallist = [goal \mid restgoals]$. An attempt is made to unify $\sigma(goal)$ with the head of a clause in the program P; to be precise the variables in the clause are first renamed to guarantee that it has no common variables with $\sigma(goal)$. Assume that a most general unifier μ exists for $\sigma(goal)$ and the head $head$ of a clause $head$:- $body \in P$, then there will be a successor node $\mathbf{n_1}$ of $\mathbf{n}$ labelled with $(body + restgoals, \mu \circ \sigma)$. Here, "+" denotes the concatenation of two lists and $\circ$ composition of substitutions. For each successful unification a successor node of $\mathbf{n}$ will be created in this way from left to right in the order of appearance in P. Branches in T_P will always be called *computations* to avoid confusion with branches in other tree structures, e.g. branches in a tableau. A computation terminates successfully if it is a finite branch in T_P and its last node $\mathbf{n_f}$ is labelled with the empty list of goals. The substitution in the label of $\mathbf{n_f}$ is called the *answer substitution* of the computation. A computation fails if its last node is labelled with $([goal \mid restgoals], \sigma)$, such that $\sigma(goal)$ is not unifiable with the head of any clause in P. The root of T_P is labelled with the initial list of goals, $goallist_0$, i.e. the query entered by the user, and the empty substitution. Since the shape of the computation tree also depends on $goallist_0$, we should strictly speak of the computation tree $T_{P,query}$ for a program P and a query $query$. We will use T_P whenever $query$ is clear from the context.

The computation tree provides only a static picture of the evaluation of a Prolog program. The dynamic behaviour is easily explained: evaluation starts with the root node of the tree. Whenever there is branching, Prolog chooses the leftmost continuation. If a computation fails, Prolog backs up to the next branching point and then continues along the leftmost continuation that has not yet been explored. This is called backtracking in Prolog terminology. If all backtracking alternatives at all branching points have been exhausted without reaching the empty list of goals the evaluation fails. Of course there is, as with all programming languages, the possibility that your program was not written carefully enough and evaluation runs into an infinite loop.

After this general description of computation trees we will take a closer look at the computation trees associated with the leanTAP program. The

root node will be labelled with the one-element goal list

```
[prove(fml,h,[],[],d)]
```

with `fml` being a formula, `h` a list of formulæ and `d` a natural number. Any node **n** with goal list

```
[prove((fml1,fml2),h,lits,freev,d) | Restgoals ]
```

will have a successor node arising from clause 1. The same goal also unifies with the head of the clauses 11 and 14 and there should be corresponding successor nodes in the computation tree. This is the right time to explain the meaning of the symbol "`!`", called cut. When "`!`" is reached during the evaluation of the body of a clause with head *head* all alternative clauses that satisfy *head* will be cut off. In the case at hand here there is consequently no branching at node **n** in the computation tree. The same remark applies to nodes labelled with

```
[prove((fml1;fml2),h,lits,freev,d) | Restgoals ].
```

The cut in clause 3 prevents branching. Note also that this time the length of the list of goals is increased. The universal quantifier case requires more explanations. Here we look at a node **n** labelled with

```
[prove(all(X,fml),h,lits,freev,d) | Restgoals]
```

and clause 6 is called. `\+length(freev,d)` succeeds if the length of the list `freev` is strictly less than the number `d`.[7] `length/2` is a built-in predicate in most Prolog systems. It may not be available in your system and you will have to program it yourself. The same may also be true for the predicate `copy_term/2`. The `append` predicate has already been mentioned above. Clause 6 is the only clause that changes the value of the fourth argument of the `prove` predicate. Because of the cut "`!`" in the body of clause 6 node **n** has only one successor node. Since the input formulæ do not contain existential quantifiers or negation signs in front of composite subformulæ it only remains to consider nodes labelled with

```
[prove(fml,h,lits,freev,d) | Restgoals ]
```

for `fml` a literal. These nodes may have two successors, the first and left-most arising from an application of clause 11 and the second arising from clause 14. We only comment on clause 11. A goal of the form `G1 -> G2` is resolved by Prolog's evaluation mechanism by first calling goal `G1`. If this fails, `G1 -> G2` also fails. In the special case of clause 11 this will never happen, see below. If `G1` succeeds then the goal `G2` is called. On backtracking only alternative solutions to `G2` are considered, `G1` is not considered again. This is crucial in the case of clause 11. Here `G1 = (fml = -Neg ; -fml = Neg)` is a disjunctive goal: first `(fml = -Neg)` is tried and only if this fails is `(-fml = Neg)` considered. If `fml` is a positive

[7] Recall that "`\+`" denotes Prologs negation as failure.

input formula	F	$=$	(F_1, F_2)
with	F_1	$=$	$(p; (q, r))$
and	F_2	$=$	$(F_{21}; F_{22})$
with	F_{21}	$=$	$(-p, -q)$
and	F_{22}	$=$	$(-p, -r)$

1 $\langle prove(F, [], [])\rangle$
2 $\langle prove(F_1, [F_2], [])\rangle$
3 $\langle prove(p, [F_2], []), prove((q, r), [F_2], [])\rangle$
4 $\langle prove(F_2, [], [p]), prove((q, r), [F_2], [])\rangle$
5 $\langle prove(F_{21}, [], [p]), prove(F_{22}, [], [p]), prove((q, r), [F_2], [])\rangle$
6 $\langle prove(-p, [-q], [p]), prove(F_{22}, [], [p]), prove((q, r), [F_2], [])\rangle$
7 $\langle prove(F_{22}, [], [p]), prove((q, r), [F_2], [])\rangle$
8 $\langle prove(-p, [-r], [p]), prove((q, r), [F_2], [])\rangle$
9 $\langle prove((q, r), [F_2], [])\rangle$
10 $\langle prove(q, [r, F_2], [])\rangle$
11 $\langle prove(r, [F_2], [q])\rangle$
12 $\langle prove(F_2, [], [r, q])\rangle$
13 $\langle prove(F_{21}, [], [r, q]), prove(F_{22}, [], [r, q])\rangle$
14 $\langle prove(-p, [-q], [r, q]), prove(F_{22}, [], [r, q])\rangle$
15 $\langle prove(-q, [], [-p, r, q]), prove(F_{22}, [], [r, q])\rangle$
16 $\langle prove(F_{22}, [], [r, q])\rangle$
17 $\langle prove(-p, [-r], [r, q])\rangle$
18 $\langle prove(-r, [], [-p, r, q])\rangle$
19 $\langle\rangle$

We have suppressed the last two arguments of the *prove* predicate since they are not relevant for propositional formulas.

Fig. 3. A successful computation for the `leanTAP` program with a propositional input formula

literal the first subgoal fails and `Neg` gets bound to `-fml`, which is the dual of `fml`. If `fml` is a negative literal, `-fml0`, then the first subgoal succeeds binding the Prolog variable `Neg` to `fml0`, the dual of `fml`. Without the Prolog-implication "`->`", backtracking would call the second subgoal and this would yield the unwanted solution `Neg` = `-fml`. Since the disjunctive goal in front of the "`->`" sign in the body of clause 11 succeeds exactly once we will not mention it in the computation tree. The leftmost successor node will instead be labelled with the goal sequence whose head is the disjunctive goal

```
(unify(Neg,L) ; prove(Lit,[],Lits,_,_))
```

where the substitution of this node binds `Neg` to the dual of the literal `fml`. If `unify(Neg,L)` succeeds then clause 11 has been successful and Prolog´s evaluation mechanism will start to work on the next goal in the sequence *restgoals*. The answer substitution is the most general unifier of

the two literals. On backtracking all possible most general unifiers between `lit` and some literal in the list of literals `lits` will be produced. If `unify(Neg,L)` fails then the clause on line (11) will recursively be called with the third argument `Lits` instead of `[Lit | Lits]`. Note also that the second argument is now set to `[]`.

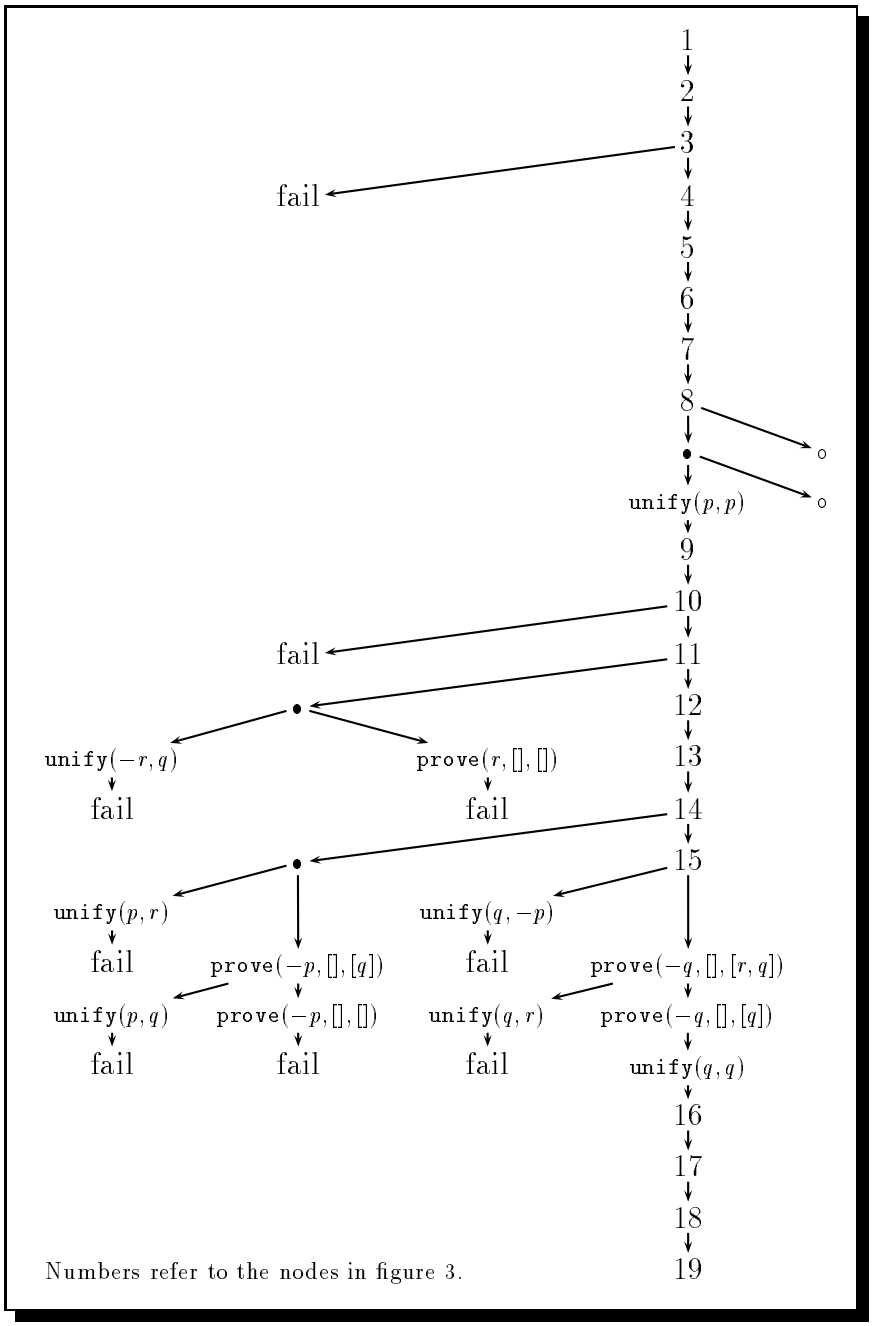


Fig. 4. The computation tree for the leanTAP program with a propositional input formula

input formula $F = (F_1, F_2, F_3)$
with $F_1 = p(a)$
and $F_2 = -p(f(f(a)))$
and $F_3 = all(X, (-p(X); p(f(X))))$

1 $\langle prove(F, [], [], [], 2)\rangle$
2 $\langle prove(p(a), [-p(f(f(a))), F_3], [], [], 1)\rangle$
3 $\langle prove(-p(f(f(a))), [F_3], [p(a)], [], 1)\rangle$
4 $\langle prove(F_3, [], [-p(f(f(a))), p(a)], [], 1)\rangle$
5 $\langle prove((-p(Y); p(f(Y))), [F_3], [-p(f(f(a))), p(a)], [Y], 1)\rangle$
6 $\langle prove(-p(Y), [F_3], [-p(f(f(a))), p(a)], [Y], 1),$
$prove(p(f(Y)), [F_3], [-p(f(f(a))), p(a)], [Y], 1)\rangle$
7 $\langle prove(p(f(a)), [F_3], [-p(f(f(a))), p(a)], [a], 1)\rangle$
8 $\langle prove(F_3, [], [p(f(a)), -p(f(f(a))), p(a)], [a], 1)\rangle$
9 $\langle prove((-p(Z); p(f(Z))), [F_3], [p(f(a)), -p(f(f(a))), p(a)],$
$[Z, a], 1)\rangle$
10 $\langle prove(-p(Z), [F_3], [p(f(a)), -p(f(f(a))), p(a)], [Z, a], 1),$
$prove(p(f(Z)), [F_3], [p(f(a)), -p(f(f(a))), p(a)], [Z, a], 1)\rangle$
11 $\langle prove(p(f(f(a))), [F_3], [p(f(a)), -p(f(f(a))), p(a)], [f(a), a], 1)\rangle$
12 $\langle\rangle$

Fig. 5. First-order example of a successful leanTAP computation

Let us consider as a specific example the leanTAP proof of $(p \vee (q \wedge r)) \rightarrow ((p \vee q) \wedge (p \vee r))$, which is just one half of the distributive law of propositional logic. The query submitted to leanTAP is $\texttt{prove}(F, [], [], X, d)$ where F is the negation of the formula to be proved, explicitely given in Figure 3. In this and the following figures of computation trees we only show goals of the form $\texttt{prove}(fml, h, lits, varlist, d)$, and sometimes also of the form $\texttt{unify}(l_1, l_2)$. A successful computation of the corresponding computation tree is shown in Figure 3. To prevent the picture from becoming too confusing we did not show the failed computations to the left of the successful path nor the branchings to the right that were not explored. This is (almost) made up for in Figure 4 on page 14. Figure 5 shows the computatation tree for leanTAP with the first-order input formula

$$p(a) \wedge \neg p(f(f(a))) \wedge \forall x(p(x) \rightarrow p(f(x)))$$

Proof. Theorem 3.0.1, part 1
By definition any computation tree is finitely branching. Thus a computation tree is finite if all its computations are finite. Let $T = T_{\mathsf{lean}TAP, query}$

for $query$ = `prove(fml,h,[],[],d)`. By $gs(\mathbf{n}) = \langle g_1, \ldots, g_k \rangle$ we denote the goal sequence attached to node $\mathbf{n}$ in T. For each individual goal g = `prove`$(fml, h, lits, varlist, d)$ we define a complexity measure $c(g)$ which is the quadruple $\langle c_1(g), c_2(g), c_3(g), c_4(g) \rangle$ of the numbers $c_i(g)$ defined below ordered lexicographically with $c_1(g)$ as the leading component.

$$\begin{array}{lcl} c_1(g) & = & d - \text{length}(varlist), \\ c_2(g) & = & \text{total number of logical connectives in } fml \text{ and } h, \\ c_3(g) & = & \text{length of } h, \\ c_4(g) & = & \text{length of } lits. \end{array}$$

Since nodes in the computation tree are labeled by goal sequences rather than single goals we need to extend the complexity measure to lists $gs = \langle g_1, \ldots, g_k \rangle$ of goals. We do this by setting $c(gs) = \langle c(g_1), \ldots, c(g_k) \rangle$ and defining a partial ordering $\prec$ on lists of quadruples of natural numbers.

Definition 3.1.1 ($\prec$).
The relation $\prec$ is the least transitive relation satisfying the following three conditions:

$$\begin{array}{lcll} \langle c^2, \ldots, c^n \rangle & \prec & \langle c^1, c^2, \ldots, c^n \rangle & \\ \langle e^1, c^2, \ldots, c^n \rangle & \prec & \langle c^1, c^2, \ldots, c^n \rangle & \text{if } e^1 < c^1 \\ \langle e^1, e^2, c^2 \ldots, c^n \rangle & \prec & \langle c^1, c^2 \ldots, c^n \rangle & \text{if } e^1 < c^1 \text{ and } e^2 < c^1 \end{array}$$

A goal sequence gs^1 is of smaller complexity than goal sequence gs^2 iff

$$c(gs^1) \prec c(gs^2)$$

The crucial observation, which is easily checked by looking at the `leanTAP` program, is that for any application of a program clause leading from the goal sequence gs^1 to the immediate successor gs^2 we have $c(gs^2) \prec c(gs^1)$. Thus finiteness of computations will follow when we can show that $\prec$ does not allow infinite descending chains. ∎

Lemma 3.1.2 (Wellfoundedness of $\prec$).
The ordering $\prec$ on lists of quadruples of natural numbers does not allow infinite descending chains.

Proof.
Assume to the contrary that an infinite descending chain $s^1 \succ \ldots s^n \succ s^{n+1} \ldots$ exists. Each element s^i in this chain is a list. Let m be the least number occuring as the length of some s^i. We must have $m > 0$ since the empty list $\langle\rangle$, which is the least element with respect to $\succ$, cannot occur among the s^i. Choose i_0 such that $s^{i_0} = \langle s^0_1, \ldots s^0_m \rangle$ has length m. We consider $s^i = \langle s^i_1, \ldots s^i_r, s^0_2, \ldots s^0_m \rangle$ and $s^j = \langle s^j_1, \ldots s^j_k, s^0_2, \ldots s^0_m \rangle$ for $j > i \geq i_0$. By choice of m and i_0 we must have $r, k > 0$. It is easily checked that all elements $s^j_1, \ldots s^j_k$ are strictly smaller with respect to $<$ than all elements $s^i_1, \ldots s^i_r$. From this it follows that the first elements in the lists

s^i with $i \geq i_0$ form a descending chain contradicting the well-foundedness of the lexicographical ordering on quadruples of natural numbers. ∎

We now turn to the proof of correctness and completeness. for both parts we will use the fact that the tableaux that can be reached from an initial set $\{fml\} \cup h$ [8] of formulæ can be retrieved from the goal sequences in the computation tree starting with $\texttt{prove}(fml, h, [], [], d)$. It turns out that this is not directly possible for all nodes. If at a node $\mathbf{n}$ execution of clause 11 is possible two successor nodes will be created, one corresponding to the $\texttt{unify}(l_1, l_2)$ predicate in the body of the clause and the other to the $\texttt{prove}(fml, [], lits, varlist, d)$ predicate. Nodes of the second type will be called *exception nodes*. At exception nodes the correspondence with the tableau structure breaks down. We first associate with every non-exception node $\mathbf{n}$ of the computation tree for a leanTAP-program the structure $tab_0(\mathbf{n})$ and secondly define a tableau structure $(tab(\mathbf{n}), \sigma(\mathbf{n}))$ for all nodes $\mathbf{n}$.

For our purposes a tableau will simply be a set of branches and a branch is simply a set of formulæ.

Definition 3.1.3 (Tableau at nodes of the computation tree). Let $\mathbf{n}$ be a node in the computation tree with attached goal sequence $gs(\mathbf{n}) = \langle g_1, \ldots, g_r \rangle$ with $g_i = \texttt{prove}(fml_i, h_i, lits_i, varlist, d)$ Then

$$\begin{array}{lcl} branch(g_i) & = & \{fml_i\} \cup h_i \cup lits_i \\ tab_0(\mathbf{n}) & = & \{branch(g_i) \mid 1 \leq i \leq r\} \end{array}$$

The definition of $tab(\mathbf{n})$ and $\sigma(\mathbf{n})$ proceed by induction on the nodes in T. For the root node $\mathbf{n}_0$ we have

$$\begin{array}{lcl} tab(\mathbf{n}_0) & = & tab_0(\mathbf{n}_0) \\ \sigma(\mathbf{n}_0) & = & \emptyset \end{array}$$

If node $\mathbf{n}_1$ is reached from node $\mathbf{n}$ with $gs(\mathbf{n}) = \langle g_1, \ldots, g_r \rangle$ by an application of the `unify` goal in the body of clause 11 then

$$\begin{array}{lcl} tab(\mathbf{n}_1) & = & tab_0(\mathbf{n}_1) \\ \sigma(\mathbf{n}_1) & = & \sigma_u \circ \sigma(\mathbf{n}) \end{array}$$

Here σ_u is the most general unifier computed in the successful execution of `unify`. If node $\mathbf{n}_1$ is an exception node reached from $\mathbf{n}$ then

$$\begin{array}{lcl} tab(\mathbf{n}_1) & = & tab(\mathbf{n}) \\ \sigma(\mathbf{n}_1) & = & \sigma(\mathbf{n}) \end{array}$$

In all other cases where node $\mathbf{n}_1$ is reached from node $\mathbf{n}$

$$\begin{array}{lcl} tab(\mathbf{n}_1) & = & tab_0(\mathbf{n}_1) \\ \sigma(\mathbf{n}_1) & = & \sigma(\mathbf{n}) \end{array}$$

[8] Strictly speaking h is a list, we assume without mentioning that h is converted into the set of its elements whenever h is used as an argument for a set theoretic operation like $\cup$.

Lemma 3.1.4. *For every node* $\mathbf{n}$ *in the computation tree of the query* $\texttt{prove}(fml, h, [], [], d)$ *there is a tableau* T *for* $\{fml\} \cup h$ *such that the tableau* $\sigma(\mathbf{n})(tab(\mathbf{n}))$ *contains all open branches of* T*.*

Proof. (of the lemma)
The proof proceeds by induction on the nodes in the computation tree and is obviously true for the root node. Let $\mathbf{n}$ be a node and T a tableau such that $\sigma(\mathbf{n})(tab(\mathbf{n}))$ contains all open branches of T. When $\mathbf{n}_1$ is reached from $\mathbf{n}$ by a program clause 1, 3 or 6 then $\sigma(\mathbf{n}_1)(tab(\mathbf{n}_1))$ contains all open branches of the tableau T_1, T_3 or T_6 respectively, where T_i is reached from T by an application of an α, β or γ rule respectively. If $\mathbf{n}_1$ is reached from $\mathbf{n}$ by program clause 14 tab and σ remain unchanged and there is nothing to show. The same is true when $\mathbf{n}_1$ is an exception node. It remains to consider the case when $\mathbf{n}_1$ is reached from $\mathbf{n}$ by the `unify` predicate in the body of program clause 11. If $g = \texttt{prove}(fml, h, lits, varlist, d)$ is the leftmost goal in $gs(\mathbf{n})$ then $tab(\mathbf{n}_1) = tab(\mathbf{n}) \setminus \{branch(g)\}$. But since $\sigma_u(branch(g))$ is closed the statement of the lemma remains true for node $\mathbf{n}_1$. ∎

Proof. (Theorem 3.0.1 Part 2)
When $\texttt{lean}T^AP$ terminates successfully at node $\mathbf{n}$ then $tab(\mathbf{n})$ is the empty set. This implies by the previous lemma that there is a tableau for the initial set of formulæ with no open branches. Thus correctness of $\texttt{lean}T^AP$ is proved. ∎

Proof. Theorem 3.0.1 (Part 3).
This part of the proof starts from the assumption that the formula F is a logical consequence of the list of formulæ H. Thus there exists a closed tableau T for $[F \mid H]$. This T may, on the face of it, not match very well with the tableau that the $\texttt{lean}T^AP$ program tries to construct. From the completeness proof of the tableau calculus (see e.g. [Fit90]) we know already more: not only does there exist a closed tableau, but **every** fair expansion strategy will eventually produce a closed tableau. It thus remains to show the tableaux $(tab(\mathbf{n}), \sigma(\mathbf{n})$ that are associated to $\texttt{lean}T^AP$'s computation tree constitutes a fair tableau search strategy.
Proof sketch:
There are only two kinds of branching points in the computation tree of $\texttt{lean}T^AP$. The first reflects the alternatives to solve a goal of the form $\texttt{prove}(lit, h, lits, varlist, d)$ where lit is a literal. Either clasue 11 or clause 14 are applicable. Clause 11 is tried first and if this does not lead to successful termination finiteness of the computation tree will force backtracking and clause 14 will be taken. This shows that for any node $\mathbf{n}$ and any branch $b \in tab(\mathbf{n})$ any formula $fml \in b$, that is not a literal, will eventually be expanded, unless the computation has in the meantime already ended successfully.

The second kind of branching points occur in the execution of the body of clause 11; there the alternative to unify *lit* with the first element in the list *lits* or to shorten the list *lits* arises. First unification is pursued. If this does not lead to successful termination finiteness of the computation tree again forces the second alternative to be considered. In this way all possibilities to close a branch will be explored.
It remains to observe that by increasing the bound d in the initial query to the leanTAP-program the number of occurences of a particular universal formula on each branch may be arbitrarily increased.
Altogether this shows that leanTAP pursues a fair search strategy. ∎

4 Including Universal Formulæ

There are many known heuristics which can be included into a tableau-based theorem prover. Including heuristics usually means either strengthening the underlying calculus for gaining shorter proofs, or directing the proof search in order to avoid useless search. Both are not universally good ideas: the fact that proof lengths decrease does not say anything on the difficulty of actually *finding* these shorter proofs, and guiding the proof search usually involves some effort for computing the particular guidelines. Heuristics are not a panacea: one must carefully analyze whether it really pays off to include a concrete heuristic. The more focussed an application of a theorem prover, the better the chances that appropriate heuristics increase the efficiency of an implementation.

From some heuristics, however, most application areas benefit and it is generally a good idea to give them at least a try. One of these domain-independent heuristics is called *universal formulæ*. The idea behind it is the following:

Universally quantified formulæ are often used more than once for closing a tableau, and each time a different substitution for the free variables is needed. The standard procedure in semantic tableaux for this is to apply the γ-rule to the corresponding formulæ more than once and generate several instances of them. Each instance contains different free variables, which can be bound differently for closing branches. The problem is that the more instances of γ-formulæ are created, the bigger the tableau (i.e. the search space) grows. Here it helps to recognize "universal" formulæ; these can be used arbitrarily often in a proof with different substitutions for some of their free variables.

Definition 4.0.1. Suppose ϕ is a formula on some tableau branch B. ϕ is *universal* with respect to the variable x if the following holds for every model $\mathbf{M}$ and every ground substitution σ:

$$\text{If } \mathbf{M} \models B\sigma, \text{ then } \mathbf{M} \models ((\forall x)\phi)\sigma.$$

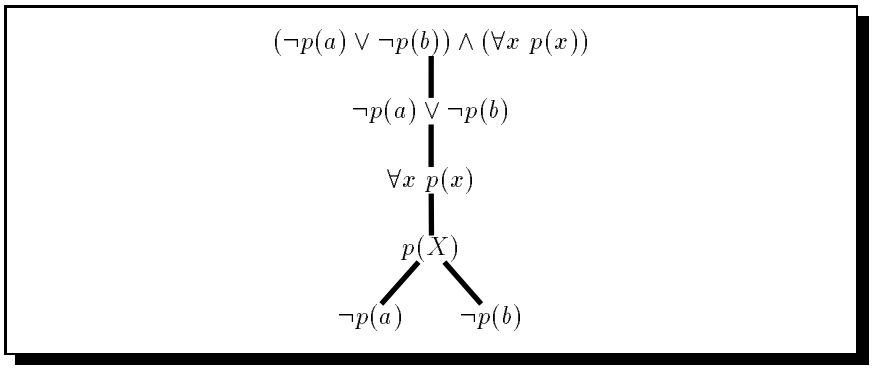


Fig. 6. An Example for the use of Universal Formulæ

Notational agreement: If we want to refer to a variable x which is universal w.r.t. a certain formula on some branch, and it is clear which branch and formula are meant, we will simply write "the universal variable x" in the sequel.

A more detailed discussion of universal formulæ can be found in [BHG+96]; we give a slightly simplified account here. Figure 6 gives an example: The variable X is universal to both branches and thus they can be closed without applying the γ-rule again.

The problem of recognizing universal formulæ is undecidable in general. However, a wide and important class of universal formulæ 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 formulæ that are generated by this sequence are universal w.r.t. the free variables introduced within the sequence. Substitutions for these variables can be ignored, since the sequence could be repeated arbitrarily often for generating new copies of these variables — without generating new branches.

Including this optimization in the previously discussed implementation is not a major undertaking: we simply collect a list of "universal" variables for each formula. For this, the arity of `prove` is extended from 5 to 7:

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

`UnivV` and `DisV` are new parameters, the use of all others remains unchanged. `UnivV` is a list of the universal variables in the current formula `Fml`. `DisV` represents something like their counterpart: it is a Prolog term containing all variables on the current branch which are *not* universal in one of the formulæ (we will call these the "disjunctive variables").[9] Each

[9] To be precise: `DisV` holds the variables which are not universal w.r.t. a formula on the current branch, whereas `UnivV` holds the variables universal w.r.t. the current

```
% Conjunction:
1  prove((A,B),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
2    prove(A,[(UnivV:B)|UnExp],Lits,DisV,FreeV,UnivV,VarLim).
% Disjunction:
3  prove((A;B),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
4    copy_term((Lits,DisV),(Lits1,DisV)),
5    prove(A,UnExp,Lits,(DisV+UnivV),FreeV,[],VarLim),
6    prove(B,UnExp,Lits1,(DisV+UnivV),FreeV,[],VarLim).
% Universal Quantification:
7  prove(all(X,Fml),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
8   \+ length(FreeV,VarLim),
9   copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
10  append(UnExp,[(UnivV:all(X,Fml))],UnExp1),
11  prove(Fml1,UnExp1,Lits,DisV,[X1|FreeV],[X1|UnivV],VarLim).
% Closing Branches:
12 prove(Lit,_,[L|Lits],_,_,_,_) :-
13   (Lit = -Neg; -Lit = Neg ) ->
14   (unify(Neg,L); prove(Lit,[],Lits,_,_,_,_)).
% Extending Branches:
15 prove(Lit,[(UnivV:Next)|UnExp],Lits,DisV,FreeV,_,VarLim) :-
16   prove(Next,UnExp,[Lit|Lits],DisV,FreeV,UnivV,VarLim).
```

Fig. 7. leanTAP with Universal Variables

unexpanded formula in `UnExp` will have the list of its universal variables attached. The Prolog functor ":" is used for this purpose.

The prover is now started with the goal

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

for showing 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 component (lines 1 and 2 in Figure 6).[10]

When dealing with disjunctions (3–6), we exploit universality of variables and rename these variables on both branches. Experiments have shown that for most examples it is best to only rename the variables in the literals. We could also rename the universal variables in `UnExp`, but this requires an extra effort which does not pay off in many cases.

Besides this, disjunctions also destroys universality: the universal variables of a disjunction are therefore not universal to its components. The

formula.

[10] The implementation given here results in both conjunctions sharing universal variables. This is correct but not necessary: the variables could be renamed in one conjunct.

```
| ?- prove((all(X,p(X)),(-(p(a));-(p(b)))),[],[],[],[],[],1) .
Call: prove((all(X1,p(X1)),(-(p(a));-(p(b)))),[],[],[],[],[],1)
Call: prove(all(X1,p(X1)),[[]:(-(p(a));-(p(b)))],[],[],[],[],1)
Call: prove(p(X2),[[]:(-(p(a));-(p(b)))],[]:all(X1,p(X1))],[],
                                                  [],[X2],[X2],1)
Call: prove((-(p(a));-(p(b))),[[]:all(X1,p(X1))],[p(X2)],
                                                  [],[X2],[],1)
Call: prove(-(p(a)),[[]:all(X1,p(X1))],[p(X2)],[]+[],[X2],[],1)
Exit: prove(-(p(a)),[[]:all(X1,p(X1))],[p(a)],[]+[],[a],[],1)
 Call: prove(-(p(b)),[[]:all(X1,p(X1))],[p(X3)],[]+[],[a],[],1)
 Exit: prove(-(p(b)),[[]:all(X1,p(X1))],[p(b)],[]+[],[a],[],1)
Exit: prove((-(p(a));-(p(b))),[[]:all(X1,p(X1))],[p(a)],
                                                  [],[a],[],1)
Exit: prove(p(a),[[]:(-(p(a));-(p(b))),[]:all(X1,p(X1))],[],
                                                  [],[a],[a],1)
Exit: prove(all(X1,p(X1)),[[]:(-(p(a));-(p(b)))],[],[],[],[],1)
Exit: prove((all(X1,p(X1)),(-(p(a));-(p(b)))),[],[],[],[],[],1)
```

Fig. 8. Trace of problem shown in Fig. 6.

tableau is split and these variables become non-universal on both resulting branches. We therefore add them to `DisV` by creating a new Prolog term[11]. Universal variables of other formulæ on the right-hand branch are renamed by `copy_term`. This allows universal variables to be instantiated differently on the two resulting branches.

When introducing a new variable by the quantifier rule (7-11), this variable becomes universal for the scope (it may loose that status if a disjunction in the scope is expanded, see above).

The next clause (lines 12–14) remains unchanged, besides having two more parameters.

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

Figure 8 shows a trace of the program when run on the example in Figure 6.

4.1 Performance

It is interesting to compare the performance of the two programs we have presented in Figures 2 and 7. Table 1 gives the respective figures for some of Pelletier's problems [Pel86]. The negated theorem has been placed in front of the axioms and the program of Figure 1 for computing negation normal form was applied as a preprocessing step.

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

Table 1. Performance of the programs given in Figure 6/Figure 7 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 `VarLim`	Branches closed	Closings tried	Time *msec*
32	3/3	10/10	10/10	10/10
33	1/1	11/11	11/11	0/10
34	??/5	−/79	−/79	∞/109
35	4/2	1/1	1/1	0/0
36	6/6	3/3	3/3	0/0
37	7/7	8/8	8/8	9/30
38	4/4	90/90	101/101	210/489
39	1/1	2/2	2/2	0/0
40	3/3	4/4	5/5	0/0
41	3/3	4/4	5/5	0/9
42	3/3	5/5	5/5	9/19
43	5/5	18/18	18/18	109/179
44	3/3	5/5	5/5	10/19
45	5/5	17/17	17/17	39/79
46	5/5	53/53	63/63	59/189

Some of the theorems, like Problem 38, are quite hard: the ${}_3T^AP$ prover [BGHK92a], for instance, needs more than ten times as long. Schubert's Steamroller (Pelletier No. 47) cannot be solved; this is no surprise, since the problem is designed for forward chaining systems based on conjunctive normal form. It can only be proven in tableau-based systems if good heuristics for selecting γ-formulæ are used. Using a queue, as in our case, is not sufficient. We console ourselves with Problems No. 34 and 38, which are barely solvable in a comparable time by CNF-based provers unless sophisticated algorithms for deriving conjunctive normal forms are applied. Pelletier No. 34 (also called "Andrews Challenge") is not solvable without universal formulæ, either. This example demonstrates the usefulness of the heuristic for complex problems. The use of universal formulæ, however, also has disadvantages: the runtime for other problems (like 38) increased, as there is some overhead involved with maintaining universal variables.[12]

[12] The overhead, however, is not dramatic: in practice, an implementation is slowed down by a constant factor of about 2. On the other hand, exploiting universal formulæ can result in an exponential speedup.

5 Tableaux as Graphs

Taken literally, the theoretical treatment of semantic tableaux seems to suggest that tableaux are trees. However, the programs presented so far as well as the completeness proof consider tableaux as sets of branches. One standard approach to improve an implementation is to look for efficient data structures. Using trees would be an impovement over sets of branches, but acyclic graphs are even better. Since graphs use structure sharing, i.e. multiple occurences of the same subtree are replaced by pointers to only one occurence of the subtree, they allow for a very compact representation of the branches of a tableau which may in extreme cases be exponentially smaller than a tree representation.

This section investigates such a graphical representation of tableaux. The underlying idea is to reduce the amount of computation required during deduction by moving some of the effort for expanding formulæ into preprocessing. The preprocessing computes a graph representation of a partially extended tableau, where α- and β-formulæ are already fully expanded and need not be considered during the proof search any more.

We begin with an explanation of the syntax used to describe graphs. The simplest tableau graph consists of one node labeled with the atom "1". This atom is used as a marker for the end of branches, i.e. it is the last entry in all branches of a tableau graph. If F and G are graphs, then $F \vee G$ will be the graph with a top node labeled with the connective $\vee$ from which two edges lead to the top nodes of graph F and graph G respectively. In addition we will use the graph constructor $\wedge$ which is particular to the class of graphs considered here. If F and G are graphs then $F \wedge G$ denotes the graph obtained from G by adding a new top node n^0 above the original top node of G. Node n^0 is labeled by F. This offers the possibility to represent graphs inside of graphs and we use it for treating universal quantifiers.

The following function maps a formula in negation normal form into a graphical representation of its partially expanded tableau:

Definition 5.0.1. (*Mapping Formulæ to Tableau Graphs*)
Let F be a first-order formula in Skolemized negation normal form, and let "1" denote the atomic truth constant[13]

$$tgraph(F) \stackrel{\text{def}}{=} \begin{cases} F \wedge 1 & \textit{if } F \textit{ is a literal} \\ tgraph(A)\left[\frac{1}{tgraph(B)}\right] & \textit{if } F = (A \wedge B) \\ tgraph(A) \vee tgraph(B) & \textit{if } F = (A \vee B) \\ (\forall x\ tgraph(F')) \wedge 1 & \text{if } F = \forall x\ F' \end{cases}$$

where

[13] W.l.g. we assume that "1" does not occur in F.

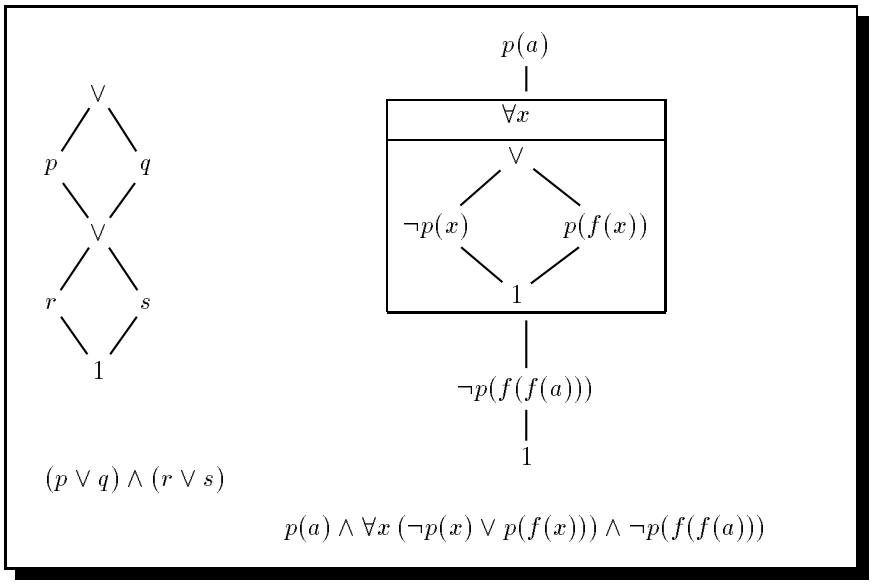


Fig. 9. Sample Tableau Graphs.

$$F\left[\frac{G}{G'}\right] \stackrel{\text{def}}{=} \begin{cases} A \wedge (B\left[\frac{G}{G'}\right]) & \text{if } F = A \wedge B \\ (A\left[\frac{G}{G'}\right]) \vee (B\left[\frac{G}{G'}\right]) & \text{if } F = A \vee B \\ G' & \text{if } F = G \end{cases}$$

The replacement function used to deal with a conjunctions $A \wedge B$ has the effect of appending the graph for B at the end of all branches of the $tgraph(A)$.

Universally quantified formulæ are represented by deriving a representation for a fully expanded tableau of the scope of the formula and putting it into one node together with a reference to the quantified variable. In this way we use nesting of graphs for treating quantified formulæ. Note that an application of the replacement mapping does never touch these nested graphs.

Figure 9 contains two examples. The left tableau graph shows the principle of structure sharing: the graph is smaller than a fully expanded tableau for the formula below, but the paths in the graph correspond to the branches of such a tableau. The graph on the right-hand side is a tableau graph with a universally quantified formula: the quantified formula is represented as a subgraph.

It may at first seem difficult to realize structure sharing in Prolog since it does not support the use of pointers. The key to the solution is to use difference lists for the implementation of the replacement operation: literally, an expression "$F\left[\frac{1}{G}\right]$" means that each occurrence of "1" is replaced

```
% Conjunction:
tgraph((A,B),GraphA/GraphEnd):-!,
  tgraph(A,GraphA/GraphB),
  tgraph(B,GraphB/GraphEnd).
% Disjunction:
tgraph((A;B),(GraphA;GraphB)/GraphEnd) :-!,
  tgraph(A,GraphA/GraphEnd),
  tgraph(B,GraphB/GraphEnd).
% Universal Quantification:
tgraph(all(X,F),(all(X,GraphF),TEnd)/TEnd):- !,
  tgraph(F,GraphF/true).
% Literals:
tgraph(Literal,(Literal,End)/End).}
```

Fig. 10. Implementing Definition 5.0.1

by an instance of G. If instead of 1 we use a Prolog variable X then assigning G to X will have the same effect. Figure 10 shows a literal translation of Definition 5.0.1 into Prolog. A goal `tgraph(Fml,G1)` will succeed with binding `G1` to a tableau graph for `Fml`. Note, that the computation of `tgraph(Fml,G1)` requires only linear effort w.r.t. to the length of the input formula `Fml`. For the propositional formula in Figure 9 the `tgraph` procedure outputs the term

$$G = (p, ((r, \_); (s, \_)); (q, ((r, \_); (s, \_))$$

which seems to duplicate the subterm $((r, \_); (s, \_))$, but this happens only when the term is printed. Internally it is represented something like

$$\begin{array}{lcllcl} G & = & X_1; X_2 & \qquad Z & = & (Z_1; Z_2) \\ X_1 & = & (p, Z) & \qquad Z_1 & = & (r, \_) \\ X_2 & = & (q, Z) & \qquad Z_2 & = & (s, \_) \end{array}$$

Figure 11 lists a prover that takes such a tableau graph as input. More precisely, the branch end markers will first be set to `true` and the difference list construct will be removed. The combination of the two programs thus looks like

```
tgraph(Fml,Graph/true) , gprove(Graph,[],[],[],d)
```

For showing that the original formula is inconsistent, we must check that each path in the represented tableau is closed. This is done by recursively descending the graph and constructing paths. A path is closed if there exists a substitution that generates a contradiction within the literals of the path. The proof search succeeds when all paths are closed.

```
1  memberunify(X,[H|T]) :- unify(X,H);memberunify(X,T).
   % Disjunction:
2  gprove((A;B),Gammas,Lits,FreeV,VarLim) :- !,
3        gprove(A,Gammas,Lits,FreeV,VarLim),
4        gprove(B,Gammas,Lits,FreeV,VarLim).
   % Noticing Universal Quantification:
5  gprove((all(X,Gr),Rest),Gammas,Lits,FreeV,VarLim) :- !,
6        gprove(Rest,[all(X,Gr)|Gammas],Lits,FreeV,VarLim).
   % Applying Universal Quantification:
7  gprove(true,[all(X,Gr)|Gammas],Lits,FreeV,VarLim) :- !,
8        \+ length(FreeV,VarLim),
9        copy_term((X,Gr,FreeV),(X1,Gr1,FreeV)),
10       append(Gammas,[all(X,Gr)],Gammas1),
11       gprove(Gr1,Gammas1,Lits,[X1|FreeV],VarLim).
   % Closing Branches:
12 gprove((Lit,Rest),Gammas,Lits,FreeV,VarLim) :-
13       (Lit = -Neg; -Lit = Neg) -> memberunify(Neg,Lits)
14              ; gprove(Rest,Gammas,[Lit|Lits],FreeV,VarLim).
```

Fig. 11. Deduction With Tableau Graphs

If a path cannot be closed, we must perform the equivalent of applying a γ-rule in a standard tableau; recall that universally quantified formulæ are represented as nested subgraphs, which contain a tableau for the scope of the quantified formula. We can simulate the application of a γ-rule by appending a copy of the subgraph to the branch we are currently considering[14]. We implement this by collecting the entry-points to such subgraphs until we end up at the end of a branch. If it is not closed, we select one of the subgraphs for expansion.

The implementation shown in Figure 11 uses the predicate

```
gprove(TGraph,Gammas,Lits,FreeV,VarLim)
```

where `TGraph` is a tableau graph, computed by the predicate `tgraph/2` as explained above. `VarLim` limits the number of free variables on every branch during the proof search (analogously to the previous programs). The other arguments, which are initially set to the empty list, represent the currently considered branch: `Gamma` will hold all universally quantified sub-tableau graphs that are valid on the current branch, and `Lits` will hold all literals on it. Free variables that have been introduced are collected in `FreeV`.

[14] In more formal notion this means: if $\forall x \Phi$ is on a branch, we conjunctively add Φ' to it, where Φ' is a copy of Φ with x being renamed. This is correct, as $(\forall x \Phi) \rightarrow \Phi'$ is valid.

The procedure for proving that the tableau graph given as input is inconsistent mainly consists of expanding the individual branches or paths in the graph. The first and the last clauses shown in Figure 11 work in a way similar to that explained in previous versions of the leanTAP program: the first clause corresponds to the implementation of a β-rule, whilst the last one closes a branch. Just for a change, we have encoded the latter clause slightly differently using a predicate `memberunify/2`: it does what its name suggests: namely implements the standard member predicate, but using sound unification.

The only part in the program that is a bit tricky are clauses 5 and 7: these implement the treatment of universally quantified formulæ. As it is a good heuristic to postpone the expansion of γ-formulæ as long as possible, we first just collect them in `Gamma` when decending a branch in the tableau graph. This is what the second clause does in a quite obvious way. Whenever we reach a leaf (denoted by `true`) of a branch, the subgraphs representing γ-formulæ come in again: program clause 7 selects the first one in the list unless the limit for free variables is reached (line 8) and replaces all free variables in the subgraph (line 9). For fairness reasons, the formula just expanded is moved to the end of the list `Gammas` (line 10), and the proof search continues with one more free variable on the branch.

The program is textually not much smaller than the previous version working with sets of branches as the data structure for tableaux, but there is less work to be carried out during the proof search. On the other hand one cannot expect too much of a speed up, since during proof search all branches have to be considered, no matter how succinctly they have been represented. We have observed a typical increase of the performance of the graphical version of about 25%. The main reason for this is that multiple expansion of formulæ is avoided and less applications of tableau rules are needed. As an example consider a formula of the type $(A \vee B) \wedge (C \vee D)$. leanTAP will expand one of the conjuncts twice. This is avoided here.

6 Compiling the Proof Search

The tableau graphs introduced in the last section provide a very compact representation of fully expanded tableaux; they can also be used for further preprocessing, like computing information about contradictory literals in advance, applying propositional simplifications, etc. One optimization we will further investigate is the *compilation* of semantic tableaux.

Compilation-based provers have been introduced by Stickel [Sti88]; the idea of this approach is to translate formulæ into executable programs that carry out the proof search during run time; it is well known that this can increase the efficiency of the proof search considerably. The reason is basically the following: rather than using a meta-interpreter that handles tableaux (or representations thereof), we compile this interpreter down into

the language we used for implementing the meta-interpreter. The result is a program that carries out the proof search for a particular set of axioms, in contrast to the meta-interpreter, which must be able to handle *any* set of axioms. This is directly comparable to interpreter and compilers in standard programming languages: Whilst an interpreter must be able to handle any program in the language, the result of compiling it is machine code for one particular program, and therefore potentially much more efficient.

```
% Auxiliary predicate that instantiates a list of variables
% to a list of integers.
%
instantiate(_,[]).
instantiate(N,[N|Tail]) :- N1 is N + 1,instantiate(N1,Tail).

tgraph(Formula,Graph) :-
        tgraph(Formula,IDs/[],Graph/(0:true)),
        instantiate(1,IDs).

tgraph((A,B),IDs/IDsTail,GrA/GrEnd):-!,
        tgraph(A,IDs/IDsB,GrA/GrB),
        tgraph(B,IDsB/IDsTail,GrB/GrEnd).

tgraph((A;B),[N|IDs]/IDsTail,(N:(GrA;GrB))/GrEnd) :-!,
        tgraph(A,IDs/IDsB,GrA/GrEnd),
        tgraph(B,IDsB/IDsTail,GrB/GrEnd).

tgraph(all(X,F),[N|IDs]/IDsT,
       (N:(all(X,GrF),GrEnd))/GrEnd) :-!,
        tgraph(F,IDs/IDsT,GrF/(0:true)).

tgraph(Literal,[N|IDs]/IDs,(N:(Literal,End))/End).
```

Fig. 12. Deriving Tableau Graphs With Labelled Nodes

Compilation-based approaches to theorem proving are usually carried out for model elimination-based calculi, only. They work by mapping formulæ in clausal form into Prolog programs, thus taking advantage of the fact that Prolog programs are Horn clauses. The resulting programs can be understood as logical variants of the clauses they have been generated for, where all contrapositives of the clauses have been created[15]. The approach presented here works differently; its principle was described in [Pos93a] and builds upon the following idea:

Instead of using a program as `gprove` for descending the graphically represented tableau, we *generate* a program that performs the search pro-

[15] Variants exist that avoid the use of contrapositives, but require a more complex deduction algorithm [BF94].

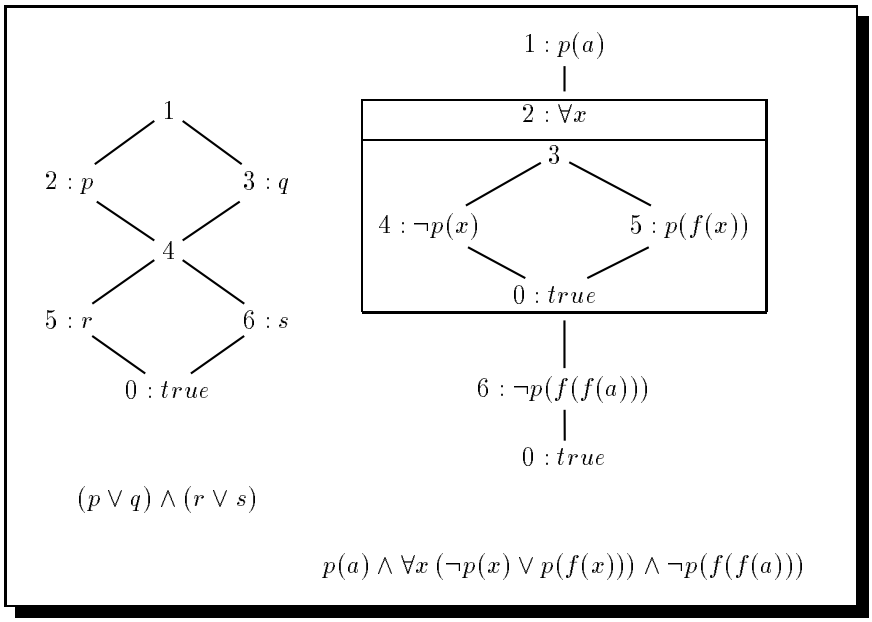


Fig. 13. Sample labeled Tableau Graphs.

cedure. Thus, we move from interpreting the graphical representation of a tableau to compiling it into a program and executing the generated code. This is the main difference from compilation-based model-elimination mentioned above: the latter transforms clauses into declaratively equivalent Prolog clauses, whereas our approach generates a procedurally equivalent Prolog program. This can, in principle, be carried out in any high-level programming language. We will, however, follow the line of this chapter and describe how to program a compiler in Prolog that generates Prolog programs.

Before we explain the idea of the compilation procedure we need a preparatory step: we have to extend the tableau-graph generation of Figure 10 by adding labels to the nodes in the graph. These will eventually serve as unique names for the generated Prolog clauses, and are necessary to control the search process and avoid duplication of Prolog clauses.

Figure 12 shows how the program for deriving tableau graphs from Figure 10 can be extended to generating graphs with labels; it works in the same way as the previous program, but additionally collects a list of Prolog variables. When the construction of the graph is completed procedure `instantiate/2` instantiates these variables to integers starting with 1. These lists of integers will serve as labels. In principle, it does not matter what sort of labels are used; integers are just a convenient choice. Note, that the label 0 is used as the label of the leaf *true*. The formulas

from Figure 9 yield the result

```
G =
   1:(2:(p,4:(5:(r,0:true);6:(s,0:true)));
   3:(q,4:(5:(r,0:true);6:(s,0:true)))
```

respectively

```
G =
  1:(p(a),2:(all(X,3:(4:(-(p(X)),0:true);5:(p(f(X)),0:true))),
  6:(p(f(f(a))),0:true)))
```

for the tgraph-procedure. A graphical representation of these outputs is shown in Figure 13.

Figure 18 lists a program that compiles labelled tableau graphs into Prolog programs. Its main predicate is `comp/3`. For a labelled tableau graph `ltgraph` the call `comp(ltgraph,X,Y)` produces a Prolog program $P_{\texttt{ltgraph}}$. Technically this is achieved by using the built-in `assert` predicate that adds the clauses of $P_{\texttt{ltgraph}}$ to the global database. The second and third arguments of `comp` are occupied by uninstantiated variables at the first call. They implement a sophisticated encoding to pass variable bindings between the generated Prolog clauses and will be explained in detail below. To explain the program in Figure 18 we have to describe the Prolog program $P_{\texttt{ltgraph}}$ it produces and the workings of the compiling program itself. It makes sense to look at the produced Prolog code first. The main predicate in $P_{\texttt{ltgraph}}$ is

```
node(+Id,+Binding,+Path,+MaxVars,+Gamma).
```

where

`Id` is the label[16] (identifier) of the corresponding node in the tableau graph.

`Binding` is a list of bindings for the variables in the tableau graph. As Prolog clauses are, by definition, variable disjoint, it is used to pass the variable bindings through the `node`-clauses at run time. The use of this parameter is a bit tricky, we will discuss it below.

`Path` is the path that has been constructed so far.

`MaxVars`, the maximal number of free variables we want to allow.

`Gamma` is a list of labels of nodes which contain universally quantified subgraphs.

The `node/5` clauses will succeed if the tableau graph below its label is inconsistent. This test is performed by considering the individual paths in the graph and by showing that all of them can be closed with a common substitution for the free variables appearing in the paths.

[16] These labels appear as the first argument, since most Prolog systems perform indexing on the first argument of a clause. Thus, the identifier is at the right position to allow fast access to clauses by their labels.

The `Binding`-parameter will be instantiated to a list: for each universally quantified variable in the initial tableau graph, there is a fixed position in the list that holds the current binding for the corresponding variable. The variable that was first encountered will always correspond to the first position in the binding list, the second encountered variable to the second position and so on. The third, forth and fifth arguments are used as in `gprove`: the argument `Path` holds the current path in the considered tableau (as a list of literals), `MaxVars` limits the number of free variables on a branch, and `Gamma` collects applicable γ-formulæ.

```
1  close(Lit,[L|Lits]) :-
2          (Lit = -Neg; -Lit = Neg) ->
3             (unify(Neg,L); close(Lit,Lits)).
4
5  start(N) :- node(1,_,[],N,[]),!.
6
7  node(0,B,P,MaxVars,[Id|Gamma]):-
8          MaxVars > 0, MaxVars1 is MaxVars - 1,
9          append(Gamma,[Id],NewGamma),
10         node(Id,B,P,MaxVars1,NewGamma).
11
12 node(1, A, B, C, D) :-
13         (   close(p(a), B)
14         ;   node(2, A, [p(a)|B], C, D)  ).
15 node(2, A, B, C, D) :-
16         node(6, A, B, C, [3|D]).
17 node(3, [_|E], A, B, C) :-
18         node(4, [D|E], A, B, C),
19         node(5, [D|E], A, B, C).
20 node(4, [D|E], A, B, C) :-
21         ( close(-(p(D)), A)
22           ; node(0, [D|E], [-(p(D))|A], B, C) ).
23 node(5, [D|E], A, B, C) :-
24         ( close(p(f(D)), A)
25           ; node(0, [D|E], [p(f(D))|A], B, C) ).
26 node(6, [D|E], A, B, C) :-
27         ( close(-(p(f(f(a)))), A)
28           ; node(0, [D|E], [-(p(f(f(a))))|A], B, C) ).
```

Fig. 14. Prolog Code for $p(a) \wedge \forall X(p(x) \rightarrow p(f(X))) \wedge \neg p(f(f(a)))$

Figure 14 gives the generated Prolog code for our running example: Lines 1–11 are not generated by the `comp`-procedure. They are part of the `node`-definition independently of the input graph. Lines 1–3 define `close`, a simple predicate which closes branches. Line 5 defines the top level predicate for starting the proof search, where the only parameter serves as a gamma limit. The clause starting in line 7 defines the action of the search

procedure when a leaf of the tableau graph is reached. If the current path can be closed the calling goal succeeds. If `MaxVars` is reached the goal fails. Otherwise the next universally quantified subgraph from the list `Gamma` is enter at its top node.

Lines 12–28 is the compiled code for the tableau graph in Figure 13: the clauses `node(1,...)` and `node(6,...)` correspond to $p(a)$ and $p(f(f(a)))$ in the graph. The universally quantified subgraph is implemented by the clause `node(2,...)`. `node(3,...)` implements the disjunction, and `node(4,...)` and `node(5,...)` correspond to the literals in the disjunct.

The trace of the program from Figure 14 for `Maxvars = 2` is shown in Figure 15

In our running example there is only one variable, this is not enough to see how the `bindings` parameter works in general. Let us consider as a second example the labelled tableau graph in Figure 16 and the prolog programm that is compiled from it in Figure 17[17].

The `binding` parameter will in this case be a three-element list which holds the bindings of the universal variables X,Y,Z of the input formula. In lines 12 and 15 of Figure 17 the first element of the binding list is accessed and used to instantitate $p(X)$ respectively $q(X)$. In lines 23 and 26 the first two positions are accessed, and the second position is used to instantiate $q(Y)$ respectively $r(Y)$. Finally in lines 34 and 37 all three positions are accessed, but only the third position is used to instantiate $r(Z)$ respectively $p(f(Z))$. The first two positions are passed on to the subsequent calls of the predicate `node`.

The open tail R avoids annecessary overhead, if only the first variable is needed the remaining positions will be lumped together in the remaining list R and passed on without change.

In line 9 of Figure 17 the Prolog variable `_` that occupies the first position in the binding list does not occur in the body of the clause. The value of `_` will not be passed on to the calls of `node(5,[A|R],X3,X4,X5)` and `node(6,[A|R],X3,X4,X5)`. Instead, a new variable A is introduced. This is exactly the effect of a γ-rule application for variable X. This also explains why the length of the `binding`-list equals the number of quantified variables in hte original tableau graph no matter how often the γ-rule is invoked. In lines 20 and 31 the same happens for the second and third variable, in our case Y and Z. Note that the values of the other positions of the binding list are passed on unchanged.

Now we have a look at the compilation program, see Figure 18 on page 37. Its main predicate is

```
comp(+TableauGraph,+BindIn,+BindOut),
```

[17] For brevity we have omitted that part of the code that does not depend on the input graph, cf. Figure 14

```
1   C: start(2)
2   C: node(1,X,[],2,[])
3   C: close(p(a),[])
3   F: close(p(a),[])
3   C: node(2,X,[p(a)],2,[])
4   C: node(6,X,[p(a)],2,[3])
5   C: close(-(p(f(f(a)))),[p(a)])
5   F: close(-(p(f(f(a)))),[p(a)])
5   C: node(0,[Y|Z],[-(p(f(f(a)))),p(a)],2,[3])
9   C: node(3,[Y|Z],[-(p(f(f(a)))),p(a)],1,[3])
10  C: node(4,[U|Z],[-(p(f(f(a)))),p(a)],1,[3])
11  C: close(-(p(U)),[-(p(f(f(a)))),p(a)])
11  E: close(-(p(a)),[-(p(f(f(a)))),p(a)])
10  E: node(4,[a|Z],[-(p(f(f(a)))),p(a)],1,[3])
26  C: node(5,[a|Z],[-(p(f(f(a)))),p(a)],1,[3])
27  C: close(p(f(a)),[-(p(f(f(a)))),p(a)])
27  F: close(p(f(a)),[-(p(f(f(a)))),p(a)])
27  C: node(0,[a|Z],[p(f(a)),-(p(f(f(a)))),p(a)],1,[3])
31  C: node(3,[a|Z],[p(f(a)),-(p(f(f(a)))),p(a)],0,[3])
32  C: node(4,[W|Z],[p(f(a)),-(p(f(f(a)))),p(a)],0,[3])
33  C: close(-(p(W)),[p(f(a)),-(p(f(f(a)))),p(a)])
33  E: close(-(p(f(a))),[p(f(a)),-(p(f(f(a)))),p(a)])
32  E: node(4,[f(a)|Z],[p(f(a)),-(p(f(f(a)))),p(a)],0,[3])
50  C: node(5,[f(a)|Z],[p(f(a)),-(p(f(f(a)))),p(a)],0,[3])
51  C: close(p(f(f(a))),[p(f(a)),-(p(f(f(a)))),p(a)])
51  E: close(p(f(f(a))),[p(f(a)),-(p(f(f(a)))),p(a)])
50  E: node(5,[f(a)|Z],[p(f(a)),-(p(f(f(a)))),p(a)],0,[3])
31  E: node(3,[a|Z],[p(f(a)),-(p(f(f(a)))),p(a)],0,[3])
. . . .
1   E: start(2)

C = Call, E = Exit, F = Fail
```

Fig. 15. Trace of the program from Figure 14

where `TableauGraph` is a tableau graph with the leaf `0:true` and `BindIn` and `BindOut` are initially the empty list.

We will refer to the binding list in the head of a `node`-clause as the *inbound binding*, and the to binding list in the body of a `node`-clause as the *outbound binding*. The compiling predicate `comp` constructs both lists while descending the tableau graph; the lists for inbound and outbound binding are passed through the calls of `comp` *without* the trailing Prolog

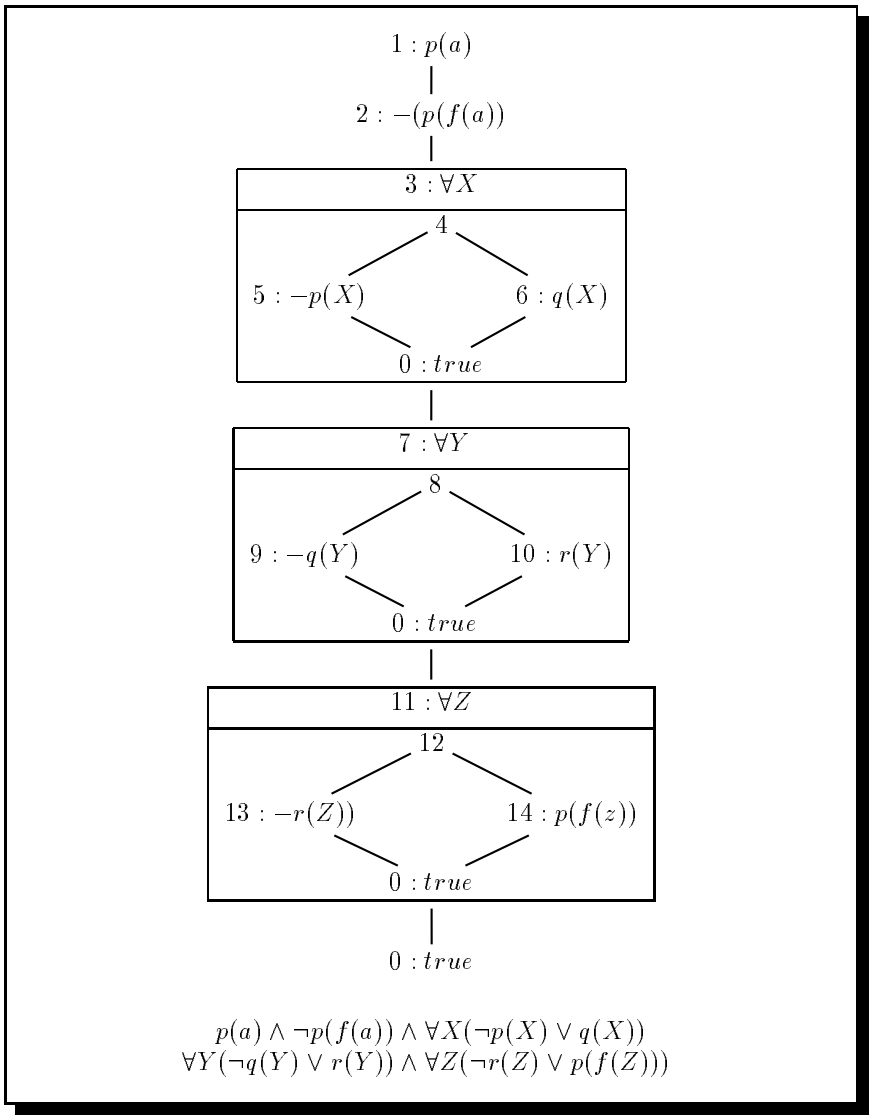


Fig. 16. Third example of a labelled tableau graph.

variable for a possible tail. The position of a variable's binding in the list is determined by the order in which the variables occur in the input graph. The only place where new variables are introduced is when compiling a universally quantified node. Then, the inbound and outbound lists are extended by one slot at their ends (lines 17,18 of Figure 18).

Line 1 terminates the recursion if a leaf is reached. The clause in line 3

```
1  node(1,X2,X3,X4,X5) :- (close(p(a),X3);
2                          node(2,X2,[p(a)|X3],X4,X5)).
3
4  node(2,X2,X3,X4,X5) :- (close(-(p(f(a))),X3);
5                          node(3,X2,[-(p(f(a)))|X3],X4,X5)).
6
7  node(3,X2,X3,X4,X5) :- node(7,X2,X3,X4,[4|X5]).
8
9  node(4,[_|R],X3,X4,X5) :- node(5,[A|R],X3,X4,X5),
10                           node(6,[A|R],X3,X4,X5).
11
12 node(5,[A|R],X3,X4,X5) :- (close(-(p(A)),X3);
13              node(0,[A|R],[-(p(A))|X3],X4,X5)).
14
15 node(6,[A|R],X3,X4,X5) :- (close(q(A),X3);
16              node(0,[A|R],[q(A)|X3],X4,X5)).
17
18 node(7,X2,X3,X4,X5) :-  node(11,X2,X3,X4,[8|X5]).
19
20 node(8,[A,_|R],X3,X4,X5) :- node(9,[A,B|R],X3,X4,X5),
21                              node(10,[A,B|R],X3,X4,X5)).
22
23 node(9,[A,B|R],X3,X4,X5) :- (close(-(q(B)),X3);
24               node(0,[A,B|R],[-(q(B))|X3],X4,X5)).
25
26 node(10,[A,B|R],X3,X4,X5) :- (close(r(B),X3);
27              node(0,[A,B|R],[r(B)|X3],X4,X5)).
28
29 node(11,X2,X3,X4,X5) :- node(0,X2,X3,X4,[12|X5]).
30
31 node(12,[A,B,_|R],X3,X4,X5) :- node(13,[A,B,C|R],X3,X4,X5),
32              node(14,[A,B,C|R],X3,X4,X5).
33
34 node(13,[A,B,C|R],X3,X4,X5) :- (close(-(r(C)),X3);
35              node(0,[A,B,C|R],[-(r(C))|X3],X4,X5)),
36
37 node(14,[A,B,C|R],X3,X4,X5) :- (close(p(f(C)),X3);
38              node(0,[A,B,C|R],[p(f(C))|X3],X4,X5)),
```

Fig. 17. Code for third example of a labeled tableau graph

succeeds if a `node`-clause with the same `id`-number has already been added. No further action will follow in this case.

Lines 6–13 compile a disjunction: in 6 and 7, we append a tail to the inbound and outbound binding and assert a `node`-clause which implements a disjunction: two goals that correspond to both generated branches must be solved. After asserting the clause, we continue to compile both disjuncts.

Lines 16–22 compile universally quantified subgraphs; we generate in-

```
1  comp(0:true,_,_):- !.
2
3  comp((Id:_),_,_) :- clause(node(Id,_,_,_,_),_),!.
4
5  % Disjunctions:
6  comp(Id:((LeftId:Left);(RightId:Right)),BindIn,BindOut):-!,
7      append(BindIn,BTail,BI),
8      append(BindOut,BTail,BO),
9      assert((node(Id,BI,P,MaxVars,Gamma) :-
10             node(LeftId,BO,P,MaxVars,Gamma),
11             node(RightId,BO,P,MaxVars,Gamma))),
12     comp(LeftId:Left,BindOut,BindOut),
13     comp(RightId:Right,BindOut,BindOut).
14
15 % Univ. quantification:
16 comp(Id:(all(X,(ScId:Scope)),SuccId:Succ),BindIn,BindOut):-!,
17     append(BindIn,[_],ScBindIn),
18     append(BindOut,[X],ScBindOut),
19     assert((node(Id,Bind,P,MaxVars,Gamma) :-
20             node(SuccId,Bind,P,MaxVars,[ScId|Gamma]))),
21     comp(ScId:Scope,ScBindIn,ScBindOut),
22     comp(SuccId:Succ,ScBindOut,ScBindOut).
23
24 % Literals:
25 comp(Id:(Lit,SuccId:Succ),BindIn,BindOut) :-!,
26      append(BindIn,BTail,BI),
27      append(BindOut,BTail,BO),
28      assert((node(Id,BI,Path,MaxVars,Gamma):-
29              close(Lit,Path)
30                ;node(SuccId,BO,[Lit|Path],MaxVars,Gamma))),
31      comp(SuccId:Succ,BindOut,BindOut).
```

Fig. 18. Compiling Tableau Graphs

bound and outbound bindings for compiling the scope of the universal quantification, as discussed above. The compiled code for the current tableau graph node does not use these: the only action we perform is to add the address of this node to the list `Gamma`, and continue by calling the node *after* the universally quantified subgraph. Thus, at runtime, we do *not* enter the universally quantified subgraph on the first transversal of the graph, but just "jump" over it ignoring its contents. The actual renaming of the quantified variable takes place if we enter the code for the subgraph during runtime: line 21 calls the compiler for this subgraph, which will generate the corresponding code. Line 22 calls the compiler for the next node after the subgraph. Note, that the changed inbound and outbound binding is only relevant for compiling these clauses. The current clause we compile does not refer to any variables, so it is sufficient to pass the

bindings with a simple Prolog variable. Here, it is called `Bind`.

The purpose of the last clause is again quite obvious: when compiling a literal, we either close the current branch or (if this fails) call the clause for the next node.

Compiling formulæ into Prolog code yields another speedup compared to the version dealing with tableau graphs: depending on the quality of your Prolog compiler, speed can easily double. An interesting point is that the compilation principle can be integrated into the "interpreting" versions of our tableau provers: One could, for instance, compile theories that are often used in advance and load the corresponding Prolog code when required.

Compilation is a powerful technique for theorem provers, but it also has its limits. It is important to understand that it does not increase the efficiency of a calculus, but "just" the efficiency of its implementation. For achieving the former we must modify the underlying calculus; one possible way is by using lemmata.

7 Including Lemmata

Lemmata will be treated extensively in another chapter of the forthcoming Handbook of Tableau-based Methods in Automated Deduction. Here, we will treat the technical aspects of including lemmata into the proof search, after having recalled the basic idea behind lemmata.

7.1 What are Lemmata?

Lemmata can be seen as one way to strengthen a tableau calculus. Although the use of lemmata can shorten proofs, this does not mean that the shorter proofs are also easier to find. In a sense, this reduces the depth of the search space for the price of broadening it. However, if we consider certain classes of theorems that have exponential proof length (like, for instance, the pigeon-hole formulæ), including lemmata is often the only way to cope with such problems.

There are two perspectives from which one can look at lemmata; one is the truth table point of view, the other is a more "operational" point of view. The former is illustrated in Figure 19: Part a) gives the truth table for the usual treatment of disjunction in a tableau calculus, Part b) for a disjunction with lemmata. Each of the two branches in a tableau resulting from expanding a disjunction corresponds to one of the shaded areas in the truth table. In a), the entry where both subformulæ are true (the bottom-right entry) is covered twice: this means, that this entry in the truth table is "covered" by two branches.

This entry is covered only once with a disjunction rule with lemmata. In terms of tableaux, this means that we add the information that the other disjunct is false to one branch when decomposing a disjunction. By

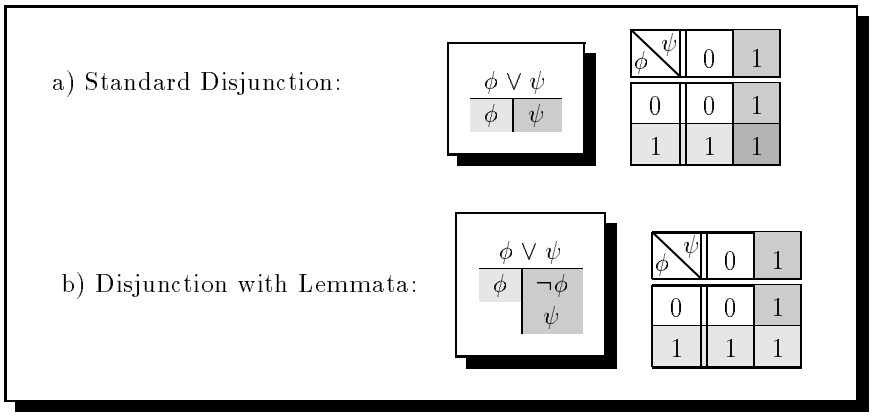


Fig. 19. Lemmata from the Truth-table Point of View.

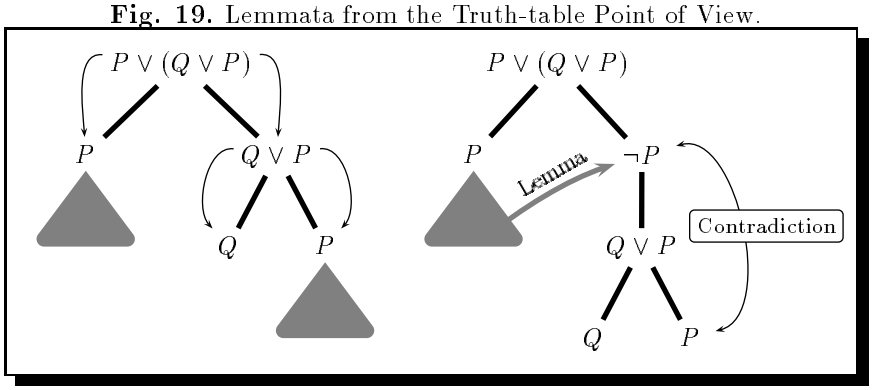


Fig. 20. Lemmata from an Operational Point of View

this, we cover each entry only on one branch. Thus, the information on branches is more specific than in the case above: no interpretation for the two disjuncts will satisfy both branches. The difference also becomes clear if we consider a fully expanded tableau for a given formula: Whilst with the standard rule for disjunctions, all paths form a DNF for the formula, we will get an XOR-normal form if lemmata are included.

Figure 20 presents this semantic consideration from another, more operational point of view: here, a lemma can be seen as a shorthand for a subproof, which would have to be carried out multiply during the proof if we did not use lemmata. As such subproofs can be arbitrarily complex, lemmata can considerably decrease the length of proofs.

7.2 Integrating Lemmata into our Framework

Lemmata are easily included into a tableau calculus by modifying the β-rules according to Figure 19: the right branch will hold the negation of the

disjunct that went to the left, i.e., we recall on the right branch that the left branch is closed.[18]. Unfortunately, this has two nasty side effects: Firstly, it is not a very good representation, as one disjunct appears twice in the expansion: We would need to represent a formula and its corresponding lemma separately. Secondly, we cannot restrict a prover to negation normal form anymore, since the required negation for building the lemma would destroy the normal form during run time.[19]

Fortunately, there are well known solutions to this problem: we can take out the screw driver and manipulate the basic representation underlying our prover: instead of building upon the disjunctive normal form-representation for tableaux, we change to an *if-then-else* representation:

Definition 7.2.1. The set of *if-then-else* expressions SH is the smallest set such that:

1. $\{0, 1\} \subset \mathrm{SH}$
2. If A is atomic and $\mathcal{B}_-, \mathcal{B}_+ \in \mathrm{SH}$ then $sh(A, \mathcal{B}_-, \mathcal{B}_+) \in \mathrm{SH}$.
3. If $\mathcal{B}, \mathcal{B}_-, \mathcal{B}_+ \in \mathrm{SH}$, then $sh((\forall x\ \mathcal{B}), \mathcal{B}_-, \mathcal{B}_+) \in \mathrm{SH}$

The semantics of $sh(A, C, B)$ is defined as: *if* A *then* B *else* C, i.e.: $(A \wedge B) \vee (\neg A \wedge C)$.

Definition 7.2.1 defines a class of formulæ which, when represented graphically, are called *BDDs* or *Binary Decision Diagrams*[20]. These formulæ are built solely by atomic formulæ, an *if-then-else*–connective and the atomic truth constants 1 and 0.

BDDs are usually defined for propositional logic, only. For handling quantifiers, we use nested *if-then-else*-expressions, analogously to tableau graphs in Definition 5.0.1. As with tableau graphs, we can easily map first-order formulæ into BDDs:

Definition 7.2.2. Let F be a first-order formula in Skolemized negation normal form; then

[18] Note, that this does not require that the left branch is actually closed before proceeding with the right branch.

[19] The latter point can be resolved if lemma generation is moved into preprocessing, e.g. if we extended the derivation of negation normal form appropriately. However, the problem of considerably increasing the size of tableaux remains.

[20] Note, that the notion "BDD" is often used in the literature to refer to ordered, reduced BDDs (ROBDDs); this is not meant here: we use non-ordered, and non-reduced BDDs. ROBDDs are considered in a later section (7.3).

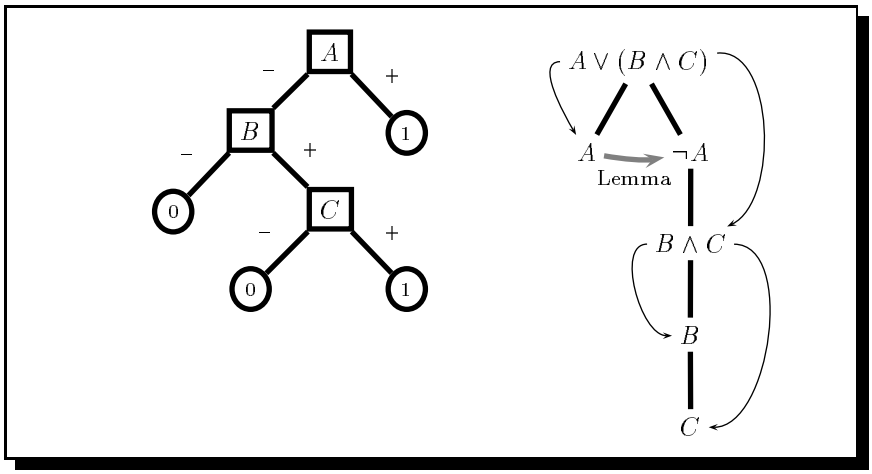


Fig. 21. BDD and Tableau for $A \vee (B \wedge C)$

$$f2Sh(F) = \begin{cases} f2Sh(A)\left[\frac{1}{f2Sh(B)}\right] & \text{if } F = A \wedge B \\ f2Sh(A)\left[\frac{0}{f2Sh(B)}\right] & \text{if } F = A \vee B \\ sh((\forall x\ f2Sh(A)), 0, 1) & \text{if } F = \forall x\ A \\ sh(F, 0, 1) \text{ or } sh(F, 1, 0) & \text{if } F \text{ is a Literal} \end{cases}$$

Figure 21 shows the BDD $f2Sh(A \vee (B \wedge C))$ and a corresponding tableaux with lemmata. If we apply the same trick as used for tableau graphs to BDDs, we can derive a graphical representation instead of a tree: the replacement operation is carried out analogously to that which was explained for Definition 5.0.1 by repacing edges, rather than nodes.

The motivation for BDDs is to handle lemmata in a better way than by modifying the β-rule; the paths to 1-leaves in BDDs are indeed nothing but a representation of branches in a tableau with disjunctive rules that incorporate lemmata. The reader is invited to verify this by examining Figure 21.

It is beyond the scope of this chapter to formalize this in detail, so we will just give the basic idea (see [Pos93b] for details), restricted to propositional logic: The key to understanding it is to compare branches in fully expanded tableaux to paths in BDDs:

Assume we have a fully expanded tableau $\mathcal{F}$ for a propositional formula F; we can then interpret the branches in $\mathcal{F}$ as conjunctions of literals. Then, the disjunction of all branches in $\mathcal{F}$ is a DNF for F.

Paths in BDDs can be regarded analogously: In propositional logic, each node in a BDD is labelled with an atomic formula. Thus, a path can be seen

as a sequence of signed atoms. The signs denote which "exit" was chosen at each node: if the *then*-part was used, the sign is positive, otherwise it is negative. Analogously to branches in tableaux, we can regard these paths as a conjunction of literals, where the sign attached to the atoms denotes whether the literal is negated or not. The difference to tableaux is that there are two kinds of paths, namely paths to 1-leafs and paths to 0-leafs. The paths to 1-leaves play the same role as the branches in tableaux, i.e., they are a DNF for the underlying formula. The 0-paths, however, build a DNF for the *negated* formula, i.e., the conjunction of all 0-paths in $f2Sh(F)$ is a DNF for $\neg F$.

To summarize, tableaux represent models of formulæ, whilst BDDs represent models *and* counter models. Note, that both our graphical representation for fully expanded tableaux (cf. Def. 5.0.1) and BDDs (Def. 7.2.2) can be computed linearly w.r.t. to the length of the negation normal form of a formula. With BDDs, we thus get the additional information of counter models more or less "for free".

It remains to show that BDDs actually fulfill their intended purpose, i.e. that they represent tableau with lemmata. We shall argue informally: Figure 20 shows how lemmata can be integrated when decomposing disjunctions in a tableau calculus; we add the negated left conjunct to the right branch. This means, the right branch will contain information about all counter models of the left disjunct when the tableau is fully expanded. The disjunction rule for computing a BDD for a formula acts similarly: for $A \vee B$, $f2Sh(B)$ is inserted for the 0-leaf of $f2Sh(A)$. As the 0-paths of $f2Sh(A)$ represent counter models of A, the 1-paths of the resulting graph will contain these. It is not very hard to show formally that the 1-paths of a BDD for a formula F are identical to the branches in corresponding tableau for F: by induction over the structure of F, we relate 1-paths to branches of a tableau for F, and 0-paths to branches of a tableau for $\neg F$. The proof is left as an exercise to the reader.

When implementing deduction based on BDDs, the first step required is to translate formulæ into the graphs. Based on the mapping given in Definition 7.2.2, we can implement this very elegantly in Prolog; Figure 22 shows a simple program which is nearly a literal translation of Definition 7.2.2:

We use the prolog *if-then-else* construct "`... -> ... ; ...`" to denote *if-then-else*. The clause

```
f2bdd(Formula,True,False,BDD)
```

succeeds if `BDD` is a BDD for `Formula` with the *true*-leaf `True` and the *false*-leaf `False`.

The first clause handles conjunctions. We recursively compute graphs for `A` and `B`, and insert the latter for the *true*-leaf of the graph for `A`. This corresponds to the first case in Definition 7.2.2.

```
1  f2bdd((A,B),True_B,False,BDD_A) :-!,
2      f2bdd(A,BDD_B,False,BDD_A),
3      f2bdd(B,True_B,False,BDD_B).

4  f2bdd((A;B),True,False_B,BDD_A):-!,
5      f2bdd(A,True,BDD_B,BDD_A),
6      f2bdd(B,True,False_B,BDD_B).

7  f2bdd(all(X,Fml),True,False,
8          (all(X,BDD_Fml) -> True; False)) :-!,
9      f2bdd(Fml,1,0,BDD_Fml).

10 f2bdd(Literal,True,False,BDD):-
11     (Literal= -Lit) -> BDD = (Lit -> False; True)
12                ; BDD= (Literal -> True; False).
```

Fig. 22. Implementing Definition 7.2.2

Disjunctions work analogously, but the graph for `B` goes to the false-leaf of the graph for `A`.

Universal quantification is handled as with tableau graphs. Note, that the leaves inside universally quantified subgraphs are instantiated to the constants `true` and `false`.

Figure 23 gives an example: it shows a graphical representation of the binding of `BDD` after successful termination of the Prolog query:

```
f2bdd((p(a),-p(f(f(a))),all(X,(-p(X);p(f(X))))),true,false,BDD)
```

7.2.1 Deduction with BDDs

From what we have seen about BDDs, it should be clear how the presented algorithms for deduction with tableau graphs can be adapted to BDDs; the underlying principle is the same: BDDs represent a disjunctive normal form and the paths in BDDs are the analog to branches in tableaux. Thus, when trying to show that a given BDD represents an inconsistent formula, we inspect its paths and try to find contradictory literals on each of them. Extension steps for applying universal quantification work in the same way as for tableau graphs. It is not difficult to modify the Prolog programs given for tableau graphs such that they work on BDDs instead.

7.3 Reduced, Ordered Binary Decision Diagrams

The reader being familiar with Binary Decision Diagrams will have noticed that we use BDDs in their non-reduced, non-ordered form[21]. In the litera-

[21] These are also called *free BDDs* by some authors, *Shannon graphs* by others.

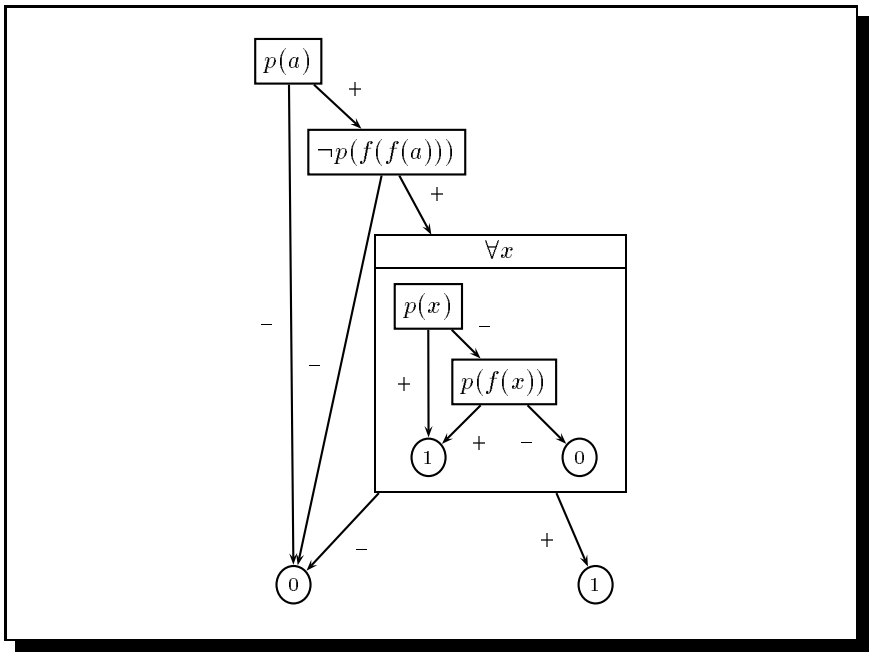


Fig. 23. An Example BDD

ture, however, BDDs appear mostly as reduced, ordered BDDs (ROBDDs) [Bry86, Bry92, GP94]. One reason for this is that BDDs are originally a propositional formalism; ROBDDs are a subclass of propositional BDDs, where

1. each path respects a given ordering on atomic formulæ,
2. no path contains multiple occurrences of the same literal, and
3. no subgraph occurs more than once in a BDD.

As a consequence of this, ROBDDs form a unique normal form for Boolean functions. ROBDDs have been successfully applied to various domains. Especially experience in hardware verification (see e.g. [BRB90]) has shown that ROBDDs are well suited as an underlying data structure for proving properties of propositional formulæ.

Our view was different: we did not use BDDs as a canonical normal form (ie: as ROBDDs), but regarded them simply as another representation of disjunctive normal forms, which is sometimes better (i.e. smaller) than tableaux. From this point of view, BDDs are nothing but a logical formulæ — possibly in a graphical representation. The logical connective underlying this representation is *if-then-else*.[22] A calculus based on BDDs

[22] Such formulæ have already been considered in 1854 by George Boole [Boo58]; Alonzo Church showed about one century later that *if-then-else* is a primitive basis for propo-

uses the same inference principles as a tableau calculus — just the underlying datastructure is different, see [GP94] for a more detailed discussion of the relation between BDDs and Automated Reasoning.

Furthermore, we used BDDs for representing first-order formulæ, and showing that they are inconsistent. This is also not a standard use of ROBDDs: these have been designed for *representing* Boolean functions, rather than for showing that the function never evaluates to 1. The main purpose of ordering atoms and maintaining a reduced graphical representation in ROBDDs is, however, to ease the representation of Boolean functions: it results in a *unique* representation (w.r.t. the ordering).

It is clear that a unique normal form for first-order logic is not computable, since the language of first-order logic is undecidable. This might appear as an argument against the use of ordered in BDDs for first-order logic, but it is not: it just says that we will not achieve a unique normal form, but does not tell anything about the efficiency of an ordered, reduced format w.r.t. the unordered BDDs we have used. The use of ROBDDs in a tableau-like setting can, from a purely logical point of view, be seen as using regular tableaux., which will be described in detail in another chapter of the forthcoming Handbook of Tableau-based Methods in Automated Deduction. It might well be the case that a first-order calculus based on ROBDDs works more efficient for a certain class of formulæ than one based on BDDs. The opposite, however, can also be the case. The answer to "what should I choose?" is not context-free.

8 A Glimpse into the Future

Automated Deduction is at present neither an engineering discipline, nor pure mathematics. The key to successfully applying Automated Deduction is careful analysis and experimenting. Both are equally important and depend on each other. The above considerations on BDDs vs. ROBDDs stress an important point in working on Automated Deduction: There is no panacea. For nearly each heuristic, or modification to a calculus, there is a counterexample where things become worse than before. The language of first-order logic is not decidable, and it is highly unlikely that this will ever change. This makes the field hard, but it also makes it interesting: we will never run out of problems.

Our motivation for writing this chapter as it is was to support experimenting: we presented a couple of implementation techniques which preserve the openess of tableau calculi. It is unlikely (although not impossible) that one of programs we presented will exactly fit for a concrete application one has. But the reader is likely to find a starting point in this chapter.

---

sitional logic [Chu56, §24, pp. 129ff].

It is hard to give any reasonable predictions of the future course of automated theorem proving in general and tableau-based automated theorem proving in particular. But it is pretty clear that the distinctions between fully automated theorem provers and interactive ones will fade. Interactive components will be added to upto now fully automated systems and automated systems will be integrated in interactive proof development systems. As one example for an interactive prover based on a sequent calculus one may name IMPS (Interactive Mathematical Proof System) described in [FGT92] and [FGT93]. Instead of implementing one prover for one logical calculus it has also been tried to develop shells that help realize a custom-made logical system. Within the family of sequent calculi such an approach has been undertaken in [RKSW94].

9 A Brief Historical Survey on Tableau-based Provers

The following list of implementations of theorem provers can certainly not claim to be exhaustive. Apart from the difficulty of locating the relevant information it is also not clear where to draw the line between tableau-based theorem provers and those that are not. We tended to include programs based on sequent calculi because of their close relationship to tableaux, but left out systems based on natural deduction. We also did not consider provers for propositional logic only. The following account is for the greatest part gleaned from [BPar].

The first tableau-based theorem prover that we know of was developed in the late fifties by Dag Prawitz, Håkan Prawitz, and Neri Voghera [PPV60]. 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.[23]

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 [Wan60]. 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[24]. 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.

[23] 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.

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

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 within a project on legal reasoning [Sch85]. 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 [Sch87]; 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 ${}_3T^AP$ system started around 1990 at Karlsruhe University [BGHK92b]; the project was funded by IBM Germany and carried out by Peter Schmitt and Reiner Hähnle. The ${}_3T^AP$ 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 bee 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 [OS88]. 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, non-monotonic logic, dynamic logic, and autoepistemic logic. Approximately at the same time a tableau-based prover was implemented at Karlsruhe University by Thomas Käufl [KZ90]; 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 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 of these workshops [FHK92, BFH+93, BDG+94].[25] Another interesting source of information on implementations are the system abstracts in the proceedings of the CADE conferences since 1986. Among the newer developments let us mention the sequent calculus based prover called GAZER [BPR92]. GAZER is implemented in Prolog.

[25] Proceedings of subsequent workshops will be published within Springer's LNCS series.

References

[BDG+94] Krysia Broda, Marcello D'Agostino, Rajeev Goré, Rob Johnson, and Steve Reeves. 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. (Workshop held in Abingdon, England).

[BF94] P. Baumgartner and U. Furbach. Model Elimination without Contrapositives. In A. Bundy, editor, *12th Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 87–101. Springer, 1994.

[BFH+93] David Basin, Bertram Fronhöfer, Reiner Hähnle, Joachim Posegga, and Camilla Schwind. 2nd workshop on theorem proving with analytic tableaux and related methods. Technical Report 213, Max-Planck-Institut für Informatik, Saarbrücken, Germany, May1993. (Workshop held in Marseilles, France).

[BGHK92a] Bernhard Beckert, Stefan Gerberding, Reiner Hähnle, and Werner Kernig. The tableau-based theorem prover ${}_3T^AP$ for multiple-valued logics. In *11th Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 758–760, Albany, NY, 1992. Springer-Verlag.

[BGHK92b] Bernhard Beckert, Stefan Gerberding, Reiner Hähnle, and Werner Kernig. The tableau-based theorem prover ${}_3T^AP$ for multiple-valued logics. In *11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science, pages 758–760, Albany, NY, 1992. Springer-Verlag.

[BHG+96] Bernhard Beckert, Reiner Hähnle, Karla Geiß, Peter Oel, Christian Pape, and Martin Sulzmann. The many-valued tableau-based theorem prover ${}_3T^AP$, version 4.0. Interner Bericht 3/96, Universität Karlsruhe, Fakultät für Informatik, 1996.

[BHS93] 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 of the 3rd Kurt Gödel Colloquium (KGC)*, Lecture Notes in Computer Science, pages 108–119, Brno, Czech Republic, 1993. Springer-Verlag.

[Boo58] George Boole. *An investigation of the laws of thought, on which are founded the mathematical theories of logic and probabilities*. Dover, New York, January1958. (First Edition 1854).

[BP94] Bernhard Beckert and Joachim Posegga. leanT^{A}P: lean,

tableau-based theorem proving. In Alan Bundy, editor, *12th Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, Nancy, France, June/July 1994. Springer-Verlag.

[BPar] Bernhard Beckert and Joachim Posegga. leanTAP: lean, tableau-based theorem proving. *J Automated Reasoning*, to appear.

[BPR92] Dave Barker-Plummer and Alex Rothenberg. The GAZER theorem prover. In *11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science, pages 726–730, Albany, NY, 1992. Springer-Verlag.

[BR93] Egon Börger and Daniel Rosenzweig. Full prolog in a nutshell. In D.S.Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, page 832. MIT Press, 1993.

[BR94] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 1994. See also [BR93].

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 40 – 45. IEEE Press, 1990.

[Bry86] Randall Y. Bryant. Graph–based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, C–35, 1986.

[Bry92] Randall Y. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical report, CMU, 1992.

[Chu56] Alonzo Church. *Introduction to Mathematical Logic*, volume 1. Princeton University Press, Princeton, New Jersey, 1956. Sixth printing 1970 .

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[Ede92] Elmar Eder. *Relative Complexities of First-Order Calculi.* Artificial Intelligence. Vieweg Verlag, 1992.

[FGT92] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: System description. In *11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science, pages 701–705, Albany, NY, 1992. Springer-Verlag.

[FGT93] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.

[FHK92] Bertram Fronhöfer, Reiner Hähnle, and Thomas Käufl. Workshop on theorem proving with analytic tableaux and related methods. Technical Report 8/92, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany, Mar1992. (Workshop held in Lautenbach, Germany).

[Fit90] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, 1990.

[GP94] Jean Goubault and Joachim Posegga. BDDs and automated deduction. In *Proc. 8th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, Lecture Notes in Artificial Intelligence, Charlotte, NC, October1994. Springer-Verlag.

[HB39] David Hilbert and Paul Bernays. *Grundlagen der Mathematik II*, volume 50 of *Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete.* Springer-Verlag, 1939.

[KZ90] Thomas Käufl and Nicolas Zabel. Cooperation of decision procedures in a tableau-based theorem prover. *Revue d'Intelligence Artificielle*, 4(3), 1990.

[OS88] F. Oppacher and E. Suen. HARP: A tableau-based theorem prover. *Journal of Automated Reasoning*, 4:69–100, 1988.

[Pel86] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.

[Pos93a] Joachim Posegga. Compiling proof search in semantic tableaux. In *Proc. 7th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, volume 671 of *Lecture Notes in Computer Science*, pages 67–77, Trondheim, Norway, June1993. Springer-Verlag.

[Pos93b] Joachim Posegga. *Deduktion mit Shannongraphen für Prädikatenlogik erster Stufe.* Infix Verlag, Sankt Augustin, Germany, 1993.

[PPV60] Dag Prawitz, Håkan Prawitz, and Neri Voghera. A mechanical proof procedure and its realization in an electronic computer. *Journal of the ACM*, 7(1–2):102–128, 1960.

[RKSW94] Bradley L. Richards, Ina Kraan, Alan Smaill, and Geraint A. Wiggins. Mollusc: A general proof-development shell for sequent-based logics. In ALAN Bundy, editor, *12th International Conference on Automated Deduction (CADE)*, volume 814 of *LNAI*, pages 826–830. Springer, 1994.

[Sch85] Wolfgang Schönfeld. Prolog extensions based on tableau calculus. In *9th International Joint Conference on Artificial In-*

telligence, Los Angeles, volume 2, pages 730–733, 1985.

[Sch87] Peter H. Schmitt. The THOT theorem prover. Technical Report 87.9.7, IBM Germany, Scientific Center, Heidelberg, Germany, 1987.

[Sti88] Mark E. Stickel. A Prolog Technology Theorem Prover. *Journal of Automated Reasoning*, 4(4):353–380, 1988.

[Wan60] Hao Wang. Toward mechanical mathematics. *IBM Journal of Research and Development*, 4(1), January1960.