Compiling Proof Search in Semantic Tableaux[*]

Joachim Posegga

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, 7500 Karlsruhe, FRG
`posegga@ira.uka.de`

Feb 8, 1993

Abstract

An approach to implementing deduction systems based on semantic tableaux is described; it works by compiling a graphical representation of a fully expanded tableaux into a program that performs the search for a proof at runtime. This results in more efficient proof search, since the tableau needs not to be expanded any more, but the proof consists of determining whether it can be closed, only. It is shown how the method can be applied for compiling to the target language Prolog, although any other general purpose language can be used.

1 Introduction

The basic idea of tableau–based systems (see (Fitting, 1990) for a good introduction) is to try to prove inconsistency of a formula by failure of a systematic model–construction process: it is tried to satisfy a formula by stepwise refinement of potential models, and a proof is found if all those models can be ruled out by detecting contradictions in them. This working principle became more and more popular in automated deduction during the last years, after resolution has governed the field for two decades. Essentially, the former proof procedures offer a closer relation to semantics than resolution does. This is handy if a deduction system is supposed to be integrated into an application, rather than designed as a stand–alone prover. The interest in this steadily increases, which explains the shift of interest.

In order to apply deduction techniques, a concrete implementation method is required: the standard way of implementing tableaux-based provers is to keep an explicit datastructure representing a tableau (usually as a set of its branches) and modify it during runtime. This can be compared with an interpreter for a programming language: the tableau is a program containing statements to be executed (i.e.: formulæ to be expanded), until certain conditions are met and the program terminates (i.e.: all branches are closed). This paper shows how the proof search can be speeded up by compilation techniques. The basic idea is to compile a fully expanded tableau into a program that carries out the proof search at runtime.

[*]This paper has been published in *Proc. Seventh International Symposium on Methodologies for Intelligent Systems*, Trondheim, Norway, June 1993 (Springer LNAI)

The underlying idea is derived from the author's work on compilation techniques for first–order deduction with Shannon graphs (Posegga & Ludäscher, 1992; Posegga, 1993) and works as follows: First, an arbitrary first–order formula is transformed into a graphical representation of a fully expanded tableau for it. Then, the graph is compiled into a program which shows the formulæ's inconsistency when it is executed. The execution reflects the proof search in semantic tableaux and tries to close every branch in the tableau. We will show how the principle works for Prolog as target language, although any other general-purpose language can be used.

The advantage of our approach is that some of the effort for the proof search (namely expanding the tableau) can be moved to a preprocessing phase that derives the graph and generates the program for it. This preprocessing is of only linear complexity in time and space w.r.t. the length of the negation normal form of the input formula. This is due to the fact that a graph instead of a tree is used for representing the fully expanded tableau; it uses structure sharing and represents multiple occurrences of subtrees in a tableau only once.

Note, that there is only a notational difference between a tableau represented as a graph and as a tree. From a theoretical point of view, trees are much handier than graphs, since we can use a linear notation and regard them simply as logical formulæ. We will refer to trees for the theoretical treatment of our method, but the reader should keep in mind that an implementation should use a graphical representation.

The paper is written from a practical point of view and assumes to be familiar with the theoretical background of semantic tableaux. When arguing on the implementation level, clearness and readability is preferred over showing how to achieve efficient code. The paper starts by discussing compilation techniques for propositional formulæ in Section 2. The framework is carried forward to the first–order level in Section 3; Chapter 4 draws conclusions from our research.

2 Propositional Logic

For reasons becoming clear soon, we will use a slightly unusual notation for writing down tableaux and consider them simply as logical formulæ with signed atoms; "+" and "−" serve as signs. Let $\mathcal{L}^0$ be the language of propositional calculus defined in the usual way, and $\mathcal{L}^0_{At}$ the atomic formulæ of $\mathcal{L}^0$.

Definition 2.1 (Set of Fully Expanded Propositional Tableaux).
The set $\mathtt{Tab}$ of fully expanded propositional tableaux is defined to be the smallest set such that

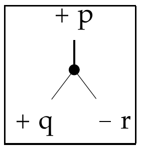
(1) $\mathbf{1} \in \mathtt{Tab}$ *("$\mathbf{1}$" denotes the atomic truth value "true")*

(2) if $\mathcal{T} \in \mathtt{Tab}$ and $A \in \mathcal{L}^0_{At}$, then $(+A)\wedge\mathcal{T}$ and $(-A)\wedge\mathcal{T} \in \mathtt{Tab}$.

(3) if $\mathcal{T}_1, \mathcal{T}_2 \in \mathtt{Tab}$, then $\mathcal{T}_1\vee\mathcal{T}_2 \in \mathtt{Tab}$.

The elements of $\mathtt{Tab}$ will be denoted by letters of the calligraphic alphabet "$\mathcal{A}, \mathcal{B}, \ldots$".

The intuition behind this notation is that the formulæ on a branch denote a conjunction, and that branching means disjunction. As a simple example, consider the formula "$p\wedge(q\vee\neg r)$"; assume we start a tableau with this formula and expand it completely in the standard way.

We get: 

which can be written as $+p\wedge((+q\wedge\mathbf{1})\vee(-r\wedge\mathbf{1}))$, an element of $\mathfrak{Tab}$.

The atom "**1**" is superfluous, but handy, as we will see soon. It can be regarded as a mark for the end of a branch.

Next, we will see how such a representation can be derived for a formula. The basic idea is to recursively compute fully expanded tableau for compound formulæ, and to combine these according to the logical connectives. The following notation will be needed for handling conjunctions:

Definition 2.2 (Replacement of **1**–nodes).
Let $\mathcal{A}, \mathcal{B} \in \mathfrak{Tab}$; the replacement of **1**–*nodes in $\mathcal{A}$ by $\mathcal{B}$ is recursively defined as:*

$$\mathcal{A}\left[\frac{\mathbf{1}}{\mathcal{B}}\right] = \begin{cases} \mathcal{B} & \text{if } \mathcal{A} = \mathbf{1} \\ \mathcal{A}_1 \wedge \left(\mathcal{A}_2\left[\frac{\mathbf{1}}{\mathcal{B}}\right]\right) & \text{if } \mathcal{A} = \mathcal{A}_1 \wedge \mathcal{A}_2 \\ \left(\mathcal{A}_1\left[\frac{\mathbf{1}}{\mathcal{B}}\right]\right) \vee \left(\mathcal{A}_2\left[\frac{\mathbf{1}}{\mathcal{B}}\right]\right) & \text{if } \mathcal{A} = \mathcal{A}_1 \vee \mathcal{A}_2 \end{cases}$$

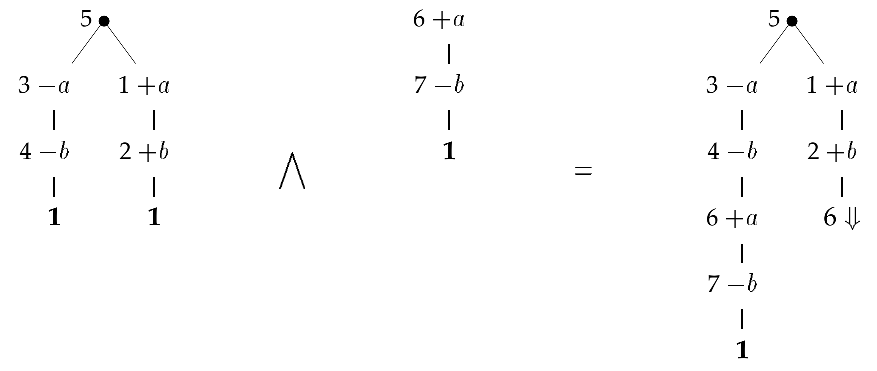
The following recursive function *f2Sh* maps an arbitrary propositional formula to a fully expanded propositional tableaux for it:

Definition 2.3.

$$f2Sh(F) = \begin{cases} (+F)\wedge\mathbf{1} & \text{if } F \in \mathcal{L}_{At} \\ (-A)\wedge\mathbf{1} & \text{if } F = \neg A, A \in \mathcal{L}_{At} \\ f2Sh(F') & \text{if } F = \neg\neg F' \\ f2Sh(\alpha_1)\left[\frac{\mathbf{1}}{f2Sh(\alpha_2)}\right] & \text{if } F \text{ is a } \alpha\text{–formula} \\ f2Sh(\beta_1)\vee f2Sh(\beta_2) & \text{if } F \text{ is a } \beta\text{–formula} \end{cases}$$

If we perform the replacement operation in the above definition by replacing edges to nodes, rather than replacing nodes themselves, we can derive a directed, acyclic graph. One easily verifies that, in this case, computing the mapping is of linear complexity in time and space w.r.t. the length of the input formula in negation normal form.

As a simple example, assume we want to derive a graph representing a fully expanded tableau for $(a \leftrightarrow b)\wedge a\wedge\neg b$. First, we compute the graph for $(a \leftrightarrow b)$, which is an β–formula with $\beta_1 = \neg a\wedge\neg b$ and $\beta_2 = a\wedge b$. After computing the graph for $a\wedge\neg b$, we conjunctively combine both by replacing all *edges* to **1**–nodes in the first graph by an edge to the second graph:



The numbers attached to each node in the graph will be needed later for the compilation. We will assume that the generated nodes are consecutively numbered, although it would be sufficient that they uniquely denote each node. The node "6 $\Downarrow$" is just an aid for drawing the graph and means that the edge leading to it actually leads to node number 6.

Besides the fact that the tableau is represented as a graph with numbered nodes, there is no difference to a standard semantic tableau. The above formula was inconsistent, so the tableaux represented by the right graph is closed, i.e.: each branch of the tableau is contradictory. In terms of the graph this means, that both paths from the root to the 1–leaf are contradictory. Once we have derived such a graph for a propositional formula, the only thing left to do for obtaining a proof is to test this condition.

This is exactly the idea of the proposed method for compiling the proof search: we compute the graph in the above way and compile it into a program that is procedurally equivalent to the above test. Any general–purpose target language can be used, but it is particularly easy to explain the process with Prolog:

As propositional logic is decidable, we can determine whether or not the graph we consider has an open path to a 1–leaf, and therefore the tableau has an open branch. If this is the case, we have derived a model for the formula. We will generate a program that enumerates all models, i.e., it enumerates all open paths in the graph. The method to achieve this is quite straightforward: for each node in the graph a Prolog clause `node/2` is generated that succeeds if an open path through this node to the 1–leaf exists. We will use the numbers of the graph nodes to distinguish the clauses for the nodes. This number is the first parameter of a `node`–clause[1], the second is the path that has been constructed to reach the node.

There are two types of nodes in a graph:

(1) binary "$\vee$"–nodes: in this case the clause for the node succeeds if an open path can be constructed through one of the successors.

(2) unary "$\wedge$"–nodes labeled with a literal: here, the clause succeeds if the literal can be added to the path without yielding a contradiction, and an open path through the successor node exists. If the successor node is 1, the last condition is always true.

[1]This is done for performance reasons, since Prolog systems usually perform indexing on the first argument of clauses.

A minor technical problem to be solved is finding an efficient representation of paths: we easily can determine the number of *different* atoms in a formula during the construction of the graph, so a good solution is to use a Prolog–term of this length. Each argument in this term represents the truth value of the according atom (denoted by "+" or "-"). The following clauses "implement" the graph for the fully expanded tableau of $(a \leftrightarrow b) \wedge a \wedge \neg b$ above; the atom a appears at the first position in the path, and b at the second. It should be easy to see that `satisfy` succeeds with a path (a model) if there is an open branch in the tableau, and fails, otherwise:

```
node(5,Path)  :- (node(3,Path) ; node(1,Path)).
node(3,Path)  :- arg(1,Path,-), node(4,Path).
node(4,Path)  :- arg(2,Path,-), node(6,Path).
node(6,Path)  :- arg(1,Path,+), node(7,Path).
node(7,Path)  :- arg(2,Path,-).
node(1,Path)  :- arg(1,Path,+), node(2,Path).
node(2,Path)  :- arg(2,Path,+), node(6,Path).
satisfy(Path) :- functor(Path,path,2),node(5,Path).
```

In the propositional case it is surprisingly easy to switch from Prolog to another target language. As we have seen, the only thing to do is to generate code that simulates descending in the graph and collects literals that form paths. This can be achieved in conventional programming languages by defining a function for each node that recursively calls functions for successor nodes after assigning a corresponding truth value to the atom of the node, if possible. Practical experiments with C and 8086 Assembler have shown, that Prolog programs of the above kind run roughly as fast as C, but about 20-30 times slower than Assembler.

3 First–order Logic

The basic difference in the proof search between propositional and first–order tableaux is that we must deal with γ–formulæ on branches in the latter case. Such formulæ have the peculiarity that they may be used more than once during the proof search. It is therefore not sufficient to determine a fully expanded tableau for a formula and try to show that it is closed. We will handle this by extending the definition of a fully expanded tableau, such that not only atoms, but also fully expanded tableaux for γ–formulæ can appear on branches. Such a node will be called a γ–node; the graph inside is called a γ–graph.

Let $\mathcal{L}$ be the language of first–order calculus and $\mathcal{L}_{At}$ the atomic formulæ of $\mathcal{L}$.

Definition 3.1 (Set of Fully Expanded First–order Tableaux).
The set $\mathcal{T}\mathtt{ab}_{\boxed{\forall}}$ of fully expanded first-order tableaux is defined to be the smallest set such that

(1) $\mathcal{T}\mathtt{ab} \subset \mathcal{T}\mathtt{ab}_{\boxed{\forall}}$

*(2) if $\mathcal{T}_1, \mathcal{T}_2 \in \mathcal{T}\mathtt{ab}_{\boxed{\forall}}$ and $x_1, \ldots, x_n$ are free variables in $\mathcal{T}_1$,
then $((\forall(x_1, \ldots, x_n)\ \mathcal{T}_1) \wedge \mathcal{T}_2) \in \mathcal{T}\mathtt{ab}$.*

The first–order counterpart of *f2Sh* will be denoted by *f2Sh*$_{\boxed{\forall}}$ and is defined as follows:

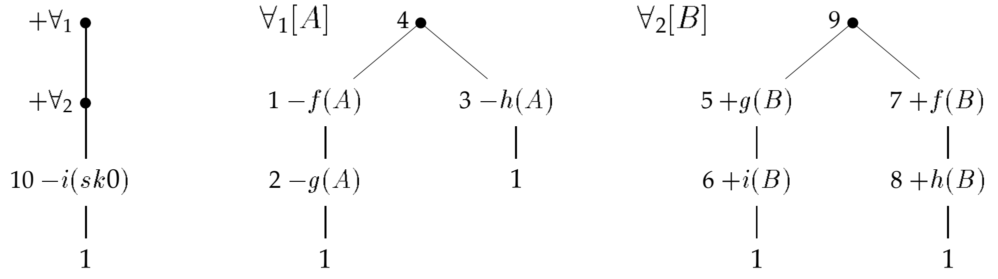


Figure 1. Tableau for Problem 3.3

Definition 3.2.

$$
f2Sh_{\boxed{\forall}}(F) = \begin{cases}
(+F)\wedge\mathbf{1} & \textit{if } F \in \mathcal{L}_{At} \\
(-A)\wedge\mathbf{1} & \textit{if } F = \neg A, A \in \mathcal{L}_{At} \\
f2Sh_{\boxed{\forall}}(F\prime) & \textit{if } F = \neg\neg F' \\
f2Sh_{\boxed{\forall}}(\alpha_1)\left[\dfrac{\mathbf{1}}{f2Sh_{\boxed{\forall}}(\alpha_2)}\right] & \textit{if } F \textit{ is a } \alpha\textit{–formula} \\
\big(f2Sh_{\boxed{\forall}}(\beta_1)\big) \vee \big(f2Sh_{\boxed{\forall}}(\beta_2)\big) & \textit{if } F \textit{ is a } \beta\textit{–formula} \\
(\forall\bar{x}\, f2Sh_{\boxed{\forall}}(F'))\wedge\mathbf{1} & \textit{if } F \textit{ is a } \gamma\textit{–formula of the form } \forall\bar{x}\ F' \\
\left(f2Sh_{\boxed{\forall}}(F' \left\{ \begin{matrix} f_1(\bar{y})/x_1 \\ \vdots \\ f_n(\bar{y})/x_n \end{matrix} \right\})\right)\wedge\mathbf{1} & \textit{if } F \textit{ is a } \delta\textit{–formula of the form } \exists(x_1,\ldots,x_n)\ F',\ \bar{y} \textit{ are the free variables in } F', \textit{and } f_1,\ldots,f_n \textit{ are new function symbols}
\end{cases}
$$

It is assumed that $\bar{x}$ stands for a finite list of variables $x_1, \ldots, x_m$. Skolemization is performed in the "liberalized δ–rule" style described in (Hähnle & Schmitt, 1991) on "$\exists$"–formulæ; substitutions are written in braces "$\{\ldots\}$".

Figure 1 shows the graph representing a fully expanded tableau for the following problem, taken from Pelletier's problem set (Pelletier, 1986):

Problem 3.3 (Pelletier 30).

Axm_pel30_1:	$\forall x\ (f(x)\vee g(x)\rightarrow\neg h(x))$
Axm_pel30_2:	$\forall x\ ((g(x)\rightarrow\neg i(x))\rightarrow(f(x)\wedge h(x)))$
Axm_pel30_3:	$\neg\forall x\ i(x)$

The left graph represents the fully expanded tableau for the whole formula, the two graphs for the γ–formulæ are drawn separately to the right of it.

Compiling such a graph for a fully expanded first–order tableau is slightly more complicated than in the propositional case. We will again explain the principle with

Prolog as the target language. First, we must choose whether we aim at proving validity or inconsistency. In principle this does not matter, but for historical reasons the latter is usually preferred. We will stick to this, simply because "reversing" things would make them less familiar and therefore less comprehensible.

For each node a clause is generated that succeeds if the paths crossing this node are contradictory. A clause for a binary node succeeds if both clauses for its successor nodes succeed, and a clause for a unary node labeled with a literal L succeeds if either

- there is a substitution σ, such that the current path and L become inconsistent under this substitution, or
- the clause for the successor node succeeds.

If we cross a γ–node for a fully expanded tableau of a γ–formula, we will just note this in the path, but we will not enter the γ–graph, yet. If we arrive with a consistent path at a `1`–leaf, one of those γ–tableaux in the path is selected and entered[2].

The basic technical problems that must be solved for the compilation process are:

(1) *Variable bindings need to be represented and passed to each clause, since Prolog clauses are – by definition – variable disjoint.*

This can be solved in the following way: when constructing the graph, all variables that appear are counted. If we have n variables, their binding can be represented by an n–ary Prolog term, each argument holding the binding for the according variable.

(2) *It is in general necessary to have more than one instance of an atom (since γ–formulæ may be used multiple), so we will not know the maximal length of a path in advance.*

For handling this we will represent a path by an open list holding signed instances of atoms[3].

(3) *γ–graphs introduce new variable bindings.*

If variable bindings are handled as pointed out in 1, the Prolog clauses for a γ–graph representing a fully expanded tableau for a γ–formula can be "re–used" arbitrarily often, if a new `binding`–term is created. The slot(s) holding binding(s) for the quantified variable(s) are simply dropped and left void by inserting an anonymous Prolog variable. So, it is possible to re–use the tableaux for γ–formulæ without asserting new clauses.

Table 1 shows the complete Prolog program for Problem 3.3. Recall that a clause succeeds if all paths crossing the according node are closed.

Each clause `node/3` "implements" one node. The first parameter is the number of the node in the graph; the second parameter is used to implement a depth–bound on the search that controls the number of applications of γ–formulæ. Paths are represented by an open list holding signed atoms, or, in the clauses "`gamma1`" and "`gamma2`" the name of the clause for the top node of a γ–graph. If the end of a path ("`1`") is reached without having found a contradiction, `use_gamma/3` selects one of them an calls the entry clause

[2]This reflects the usual treatment of γ–formulæ in tableaux: they are not used if the branch can be closed immediately.

[3]There are of course more efficient solutions than this, but we will prefer readability over efficiency, here.

```
use_gamma(0,_,_):- !,fail.
use_gamma(Limit,Path,VarBnd) :-
  NewLimit is Limit - 1,
  member(gamma(N),Path),
  node(N,NewLimit,Path,VarBnd).

close(+L1,[H|Path]) :-
  (H = -L2, unify(L1,L2)); close(+L1,Path).
close(-L1,[H|Path]) :-
  (H = +L2, unify(L1,L2)); close(-L1,Path).

node(gamma1,Limit,Path,VarBnd):-
  node(gamma2,Limit,[gamma(4)|Path],VarBnd).

node(gamma2,Limit,Path,VarBnd):-
  node(10,Limit,[gamma(9)|Path],VarBnd).

node(1,Limit,Path,bind(A,B)):-
  (close(-f(A),Path) ;node(2,Limit,[-f(A)|Path],bind(A,B))).
node(2,Limit,Path,bind(A,B)):-
  (close(-g(A),Path) ;use_gamma(Limit,[-g(A)|Path],bind(A,B))).
node(3,Limit,Path,bind(A,B)):-
  (close(-h(A),Path) ;use_gamma(Limit,[-h(A)|Path],bind(A,B))).
node(4,Limit,Path,bind(_,B)):-
  node(1,Limit,Path,bind(A,B)),
  node(3,Limit,Path,bind(A,B)).
node(5,Limit,Path,bind(A,B)):-
  (close(+g(B),Path) ;node(6,Limit,[+g(B)|Path],bind(A,B))).
node(6,Limit,Path,bind(A,B)):-
  (close(+i(B),Path) ;use_gamma(Limit,[+i(B)|Path],bind(A,B))).
node(7,Limit,Path,bind(A,B)):-
  (close(+f(B),Path) ;node(8,Limit,[+f(B)|Path],bind(A,B))).
node(8,Limit,Path,bind(A,B)):-
  (close(+h(B),Path) ;use_gamma(Limit,[+h(B)|Path],bind(A,B))).
node(9,Limit,Path,bind(A,_)):-
  node(5,Limit,Path,bind(A,B)),
  node(7,Limit,Path,bind(A,B)).
node(10,Limit,Path,VarBnd):-
  (close(-i(sk0),Path) ;use_gamma(Limit,[-i(sk0)|Path],VarBnd)).

prove(Limit) :- node(gamma1,Limit,[],_).
```

Table 1. Prolog Program for the Tableau of Figure 1

unless a depth bound is reached. `close/2` tries to find a substitution such that a path is closed. Note, that this predicate must enumerate all substitutions during backtracking.

A proof is done by calling the top node (`gamma1`) with the empty path. `proof(2)` succeeds, whereas `proof(1)` fails.

4 Conclusion & Outlook

We have described an approach to tableaux–based theorem proving that works by translating an arbitrary first–order formula into graph representing a fully expanded tableau for it. This graph can then be compiled into a program that performs the proof search when executed. We showed how to do this with Prolog as a target language, although any other general–purpose language can be used.

The principle of compiling formulæ into a general–purpose target language offers several advantages; inter alia,

- it can considerably speed up the proof search,
- it offers a flexible way for embedding deduction into applications, since the compilation can generate stand–alone subroutines that do not require a logical engine to run on.

The presented Prolog code can also be further optimized, especially the first–order version. The propositional version seems to be not very far from an optimum, apart from the fact joining clauses that have only one caller ("unfolding" the Prolog code) might result in more efficient Prolog code.

The method differs from other approaches to deduction by Horn–clause generation (e.g. (Stickel, 1988)), in that

(1) the generated program has no direct logical relation to the formula that is to be proven (i.e., the Prolog clauses are not a logically equivalent variant of the formula), but that they are *procedurally* equivalent to the search for a model.

(2) The method does not require a conjunctive normal form.

(3) The cost for translating a formula to the proposed representation of a fully expanded tableau, and the compilation of the tableau are both linear (in space an time) w.r.t. the length of the input formula.

One can argue that the proof search in a tableau–based prover usually works by stepwise developing it until each branch in it is closed; therefore, it might happen that branches close with some compound formulæ they are holding. Considering only fully expanded tableaux, as this approach proposes, results in a tableau proof procedure that never closes a branch in this way, but with literals only. From a theoretical point of view this might prevent finding a short proof; in practice, however, it very rarely happens that branches can indeed be closed with compound formulæ. So, it seems justified to neglect this issue.

An experimental implementation of a variant of the method (see (Posegga & Ludäscher, 1992)) has shown the the speedup gained by generating a Prolog program instead of using a "traditional" approach to implementing a tableau prover (without compilation) in Prolog is around a factor of 10. Using Assembler as the target language (which has been implemented for a propositional version) results in a program that runs another 20–30 times faster than the version compiling to Prolog.

References

Melvin C. Fitting. *First–Order Logic and Automated Theorem Proving*. Springer, New York, 1990.

Reiner Hähnle & Peter H. Schmitt. The liberalized δ–rule in free variable semantic tableaux. *Journal of Automated Reasoning (to appear)*, 1991.

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

Joachim Posegga & Bertram Ludäscher. Towards first-order deduction based on shannon graphs. In *Proc. German Workshop on Artificial Intelligence*, LNAI, Bonn, Germany, 1992. Springer.

Joachim Posegga. Deduktion mit Shannongraphen für Prädikatenlogik erster Stufe. Dissertation, Universität Karlsruhe, FRG, 1993.

Mark E. Stickel. A Prolog Technology Theorem Prover. In E. Lusk & R. Overbeek, editors, *9th International Conference on Automated Deduction*, Argonne, Ill., May 1988. Springer-Verlag.