(Ab)using ClAM for proving theorems in tableau calculus

Joachim Posegga

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe
7500 Karlsruhe, FRG
Email: Posegga@fra.uka.de

Abstract

This note describes an application of ClAM as a tableau based theorem prover for first order logic. Tableau rules are modeled as methods, and branches of a tableau as “sequents” that are manipulated by these methods. The resulting branch(es) are posted as subgoals. Additionally there is a method for detecting closed branches, i.e., solved subgoals.

1 Introduction

Using a system like ClAM to plan inferences in a classical first order refutation prover is certainly an interesting undertaking. However, it hard to see how this can be done in uniform proof procedures like resolution. The most powerful systems in this area, like otter [McCune 88] and PTTP [Stickel 88] use rather inflexible, but efficiently implemented inference strategies. When trying to apply planning techniques, the first thing that had to be done is making these search strategies more flexible in order to guide the search by a plan. However this would mean to give up the principle the prover’s power is based on. Additionally, resolution seems to be not the best choice when thinking in terms of planning, because it can be seen as trying to find a plan in a world that can be manipulated by only one action. A rather odd situation in planning.

It seems more reasonable to switch to a calculus with a broader range of inference rules, as tableau calculus for instance. Analytic tableaux go back to Beth and Smullyan [Beth 86, Smullyan 88], for a gentle introduction see Fitting’s book [Fitting 83]. Tableau calculus is quite popular for implementing non-classical logics, but is used seldom for “ordinary” first order logic. (Implemented theorem provers are described in [Schmitt 87, Oppacher & Suen 88]).

This note describes some experiments on applying ClAM to tableau calculus. The basic idea is easily explained: ClAM methods describe tableau rules, and an initial branch of a tableau is given as a goal. The planner tries to find applicable methods that manipulate branches and post subgoals. A certain method detects closed branches and “eats” up subgoals.

Notes in this series are for baked ideas, for \( \epsilon \geq 0 \). Only exceptionally should they be cited or distributed outwith the Mathematical Reasoning Group.

Author’s current address: Dept of AI, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland, Email: joachim@aipna.edinburgh.ac.uk
2 Tableaux

The basic principle of analytic tableaux is proving a formula by failure of constructing a falsifying model for it. This means, a proof for $\phi$ is done by trying to construct a model for $\neg \phi$. If there is no way to construct such a model, then $\neg \phi$ is unsatisfiable and hence, $\phi$ is proved.

The procedure can best be described by referring to trees: if it is to be shown that $\phi$ has no model, then $\phi$ as assumed to to be the root of an initial tree. The tree is then grown by replacing it’s nodes with the result of applying certain rules to these nodes. The rules basically decompose the formulas towards an atomic level, by preserving satisfiability. A branch of such a tree is said to be “closed” if it holds two complementary nodes, which means that the formulas on the branch cannot be satisfiable. A tableau is called closed, if all it’s branches are closed. If this is achieved, it is proven that the initial formula is unsatisfiable.

To reduce the number of rules required, only formulas in negation normal form will be considered, i.e. the only connectives used are $\lor$ and $\land$, and all negations must be at atomic level. Under this restriction, only four rules for manipulating nodes are required, given in figure 1. The rules are to be understood as follows: replace a formula of the type shown above the line by the ones shown below. A "$^*$" indicates that the tree branches.

\[\begin{array}{ll}
\text{(\land-elimination)} & \phi \land \psi \quad \rightarrow \quad \frac{\phi}{\psi} \\
\text{(\lor-elimination)} & \phi \lor \psi \quad \rightarrow \quad \frac{\phi}{\psi} \\
\text{(\forall-elimination)} & \forall x; \phi \quad \rightarrow \quad \frac{\phi^x_X}{\forall x: \phi} \\
\text{(skolemization)} & \exists x; \phi \quad \rightarrow \quad \frac{\phi^*}{\phi^*_t}
\end{array}\]

where $\phi^x_X$ is the result of replacing every occurrence of $x$ in $\phi$ by a free (meta-)variable that does not appear in the tableau.

where $\phi^*_t$ is the result of every occurrence of $x$ in $\phi$ by a term $t(X)$, $X$ are all free variables occurring in $\phi$, and $t$ is a constant new to the tree. $^*$

Note that the $\gamma$ and $\delta$ rules differ from the standard ones (e.g. those given in [Smullyan 68, Beth 86]). They are more implementation oriented and have been proposed in [Schmitt 87]. The standard $\gamma$-rule allows substituting the complete (possibly infinite) Herbrand universe for the quantified variable. This is rather inefficient, because it is hard to guess which term will be actually needed for closing a branch. The idea is to substitute a meta-variable and change the condition for a branch to be closed accordingly: a branch is closed, if it holds a pair of complementary formulas that unify under a substitution compatible with those used to close other branches. Additionally, the $\delta$-rule must be changed, it has to skolemize the formula, then.

3 The Methods

In order to use CLAM for finding a plan of a tableau proof, the most straightforward way is to regard the tableau rules as tactics and write a couple of methods that specify these tableau rules. Then, the initial set of formulas (theorem negated) is represented as a branch, which can simply be a Prolog list of formulas. This initial list is fed into CLAM, pretending that the Prolog list is a sequent (it won’t realize that it isn’t).

The planner checks the method’s preconditions, and, if one succeeds, it’s postconditions generate a branch (output sequent) that would result from applying the according tableau rule to the
incoming branch\textsuperscript{3}. The planner then tries recursively to solve the subgoal(s) (resulting branch(es)). All you need additionally is some way of realizing that a branch is closed to "eat up" goals. This can also be done by a method that is always checked first.

Let's be more concrete: figure 2 shows the methods for closing a branch and the tableau rules. The way formulas are represented in Prolog should be obvious. The logical connectives are -, \&, and \(\forall x \phi\) is written as \texttt{all/ex([x], \phi)} (note that \(x\) is a Prolog constant).

### Method for Closed Branch

```
method(closed(Branch),
        Branch,
        [member(A,Branch,Rest_br),
         copy_term(A,C_A),
         member(-C_A,Rest_br,-),
         unify(C_A,A)],
        [], [],
        closed(Branch)).
```

### Method for Alpha Rule

```
method(alpha(Branch,A & B),
        Branch,
        [member(A & B,Branch,Rest_br)],
        [append(Rest_br,[A,B],New_br)],
        [New_br],
        alpha(Branch,Formula)).
```

### Method for Delta Rule

```
method(delta(Branch,ex([Z],F)),
        Branch,
        [member(ex([Z],F),
                 Branch,Rest_br)],
        [skolemized(Z,F,Sk_Formula),
         append(Rest_br,[Sk_Formula],New_br),
         [New_br],
         beta(Branch,Formula)).
```

### Method for Gamma Rule

```
method(gamma(Branch,all([X],F)),
        Branch,
        [member(all([X],F),Branch,Rest_br)],
        [replace_var(X,F,NewF),
         append(Branch,[NewF],New_br)],
        [New_br],
        gamma(Branch,all([X],F)).
```

### Method for Beta Rule

```
method(beta(Branch,A \lor B),
        Branch,
        [member(A \lor B,Branch,Rest_br)],
        [append(Rest_br,[A,B],New_br1),
         append(Rest_br,[A],New_br2)],
        [New_br1,New_br2],
        beta(Branch,Formula)).
```

### Reminder

```
method(<name>,
        <input sequent>,
        [<precondition>,...],
        [<postcondition>,...],
        [<output sequent>,...],
        <tactic>).
```

**Figure 2: Methods for Tableau Rules**

Besides these methods, only two more things are needed to make it work:

1. writing the predicates 'replace_var' and 'skolemized' which is a fairly easy exercise, and
2. control the application of \(\gamma\)-methods.

The problem is the following: Assume the initial branch is \texttt{[all([x],p(x))]}, then the \(\gamma\)-rule is infinitely often applicable and as it is the only rule applicable, it will produce \texttt{[all([x],p(x)),p(x)]}.

To handle this, the \(\gamma\)-method that is actually used keeps track how often it applies to a certain formula and the user can specify an upper bound.

Additionally, the implementation does some more things, like giving names to branches and nodes, counting inferences, producing human readable output on demand, etc. But this is done for technical reasons only and doesn't contribute to finding a plan. The above things are basically what's needed.

The only problem with CL\textsc{AM} itself was that it has a certain optimization in its depth first planner that commits plans once they are successful. This means it never bothers to find an alternative for a terminating method that can be applied. The only terminating method here

\textsuperscript{3}The method for the \(\beta\)-rule generates two output sequents, as the rule makes the tree branch.
is the “close”-method. However, closing a branch possibly instantiates variables, and this might prevent closing of future branches (as will be seen in section 4).

4 A Sample Proof

With the above scenario the tableau prover can be used, and here’s a sample proof. It is admittedly really trivial, but shows the principle.

Assume we want to refute the following formula: \( F = \neg p(a) \land \neg p(b) \land \neg q(b) \land \forall x: p(x) \lor q(x) \).

In order to save a couple of applications of \( a \)-rules, we initialize the tableau with a branch holding the nodes

\[ \{ \neg p(a), p(b), q(b), \forall x: p(x) \lor q(x) \} \]

. The methods are loaded in the following order: close, \( a \), \( \delta \), \( \beta \), \( \gamma \).

Let’s first have a look at the depth first proof. Figure 3 shows how the tableau is grown by each successful step. The gamma limit is set to 2.

0) This is the initial branch fed into \( \text{CIAM} \). The only method applicable here is the \( \gamma \)-method.

1) The \( \gamma \)-formula \( d \) has been expanded into \( d11 \) by inserting a free (Prolog) variable. \( d \) has been labeled with a ’1’ saying that a \( \gamma \)-rule has been applied to it once.

2) Two rules could have been applied to the previous branch (\( \gamma \) and \( \beta \)), as the \( \beta \)-rule comes first, the depth first planner uses it first. After that the close-method has been used to close the branch by unifying \( p(X) \) with \( \neg p(a) \).

3) If the last branch had been closed with \( p(X) \) and \( \neg p(b) \) instead of \( \neg p(a) \), this would have been the end of the proof, as the branch left would close now as well. Unfortunately \( \neg p(a) \) was choose (simply because it comes first on the branch), and the unifying substitution \( \{ X \leftarrow a \} \), which must be applied to the whole tableau, instantiates \( X \) in \( d12 \) to \( a \). This make it impossible to close the other branch.

Here, the only thing that can be done is choosing a \( \gamma \)-extension producing \( d2 \). \( d \) has been labeled with ’2’ as it was used for two times.

Note that this situation is very similar to 1).

4) The new formula \( d22 \) is expanded by a \( \beta \)-method\(^4\).

The first thing \( \text{CIAM} \) does next is closing the branch with the leaf \( d21 \). It does so by unifying \( d21 \) with \( a \) and instantiating \( Y \) to \( a \). However, after doing that, it fails to find any applicable method. So, \( \text{CIAM} \) backtracks and finds another way of closing the above branch, ie unifying \( d21 \) with \( b \) and instantiating \( Y \) to \( b \). Then the last branch closes also and the plan (proof) is complete.

As it can be seen, the gamma limit is really important for finding the plan in a depth first way and has a strong influence on the length of plans. Figure 4 shows the output of \( \text{CIAM} \)’s proof plans found by the depth first, and the breadth first planner. When going breadth first things are different, always the shortest plan is found and a gamma limit is superfluous.

5 Is it proof planning, or theorem proving?

First thing to notice is that there is no difference between applying a method during plan search and actually executing an inference step (tableau rule, or tactic). I would guess that this does not only apply to the particular methods proposed here, but is true in general (remember: I’m talking about doing refutation proofs in first order logic, not about the domain \( \text{CIAM} \) is usually working in). It is hard to see how a method can describe the effects of a tableau rule completely unless it does something at least as expensive as applying the rule.

\(^4\)This was the only choice, because \( d \) has already been used two times (the gamma limit given to the prover).
So, if there is exactly one method for each rule, planning equals to theorem proving, and the plan in fact is the proof. In this sense, it is not planning, but proving. However, it should be easy to come up with new methods that describe combinations of tableau rules and move closer towards planning. Some good starting points are given in [Murray & Rosenthal 89] or [Oppacher & Suen 88].

6 Concluding Remarks

The implementation of a tableau prover by CLAM methods as described above has some obvious disadvantages:

1. it is probably less efficient than a “pure” implementation of a tableau prover,
2. the way universally quantified variables are treated is not the ultimate solution, and
3. the order of the methods in the database has a strong influence on the search space.

However, there are also some nice things to say about:

1. It is complete\(^5\) and sound.
2. It is easily implemented.
3. It is quite flexible in terms of adding new methods on top of the basic ones. These could be “tuned” for a certain application where for instance the axiom set is fixed, and only the theorem to be proved differs.
4. It is a good starting point for investigating proof plans in the domain of refutation proofs in classical logic.

Some ideas for improvement are (besides combining methods to more complex ones):

1. writing a procedure for detecting loops in the depth first planning process. Situation 1 and 3 in figure 3 are such an example: there is only one open branch, and both branches hold the same formulas (up to variable renaming).
2. extend CLAM to plan backwards: assume you have a database of certain tableaus that are known to be closable by some sequence of rules. Is there a way to map the current branch to one that matches these?
3. Find a way for improving the $\gamma$-rule. Unfortunately this will (probably) require gaining control of variable binding during a proof, which is a pain to implement in Prolog\(^6\).
4. Make the method for closing branches behave more clever, to avoid backtracking, which is really expensive in Prolog.

References


\(^5\)This does not apply to the depth first planner, because of the limit set on applying a gamma rule.

\(^6\)In fact Prolog III offers some features that should support that.


Figure 3: Sample Proof
[Initializing ... ]
gamma_limit(2)
axiom ~p(a)
axiom ~q(b)
axiom ~p(b)
axiom all([x], p(x)v q(x))

[Starting Proof Planner (depth first) ... ]
[Plan found after 1184 msec CPU-time]
gamma(branch(1), all([x], p(x)v q(x))) then
  beta(branch(1), p(a)v q(a)) then
    [close(branch(1), 'd1.1', p(a)),
     gamma(branch(3), all([x], p(x)v q(x))) then
      beta(branch(3), p(b)v q(b)) then
        [close(branch(3), 'd2.1', p(b)),
         close(branch(5), 'd2.2', q(b))]
  ]

[Initializing ... ]
gamma_limit(2)
axiom ~p(a)
axiom ~q(b)
axiom ~p(b)
axiom all([x], p(x)v q(x))

[Starting Proof Planner (breadth first) ... ]
[Plan found after 484 msec CPU-time]
gamma(branch(1), all([x], p(x)v q(x))) then
  beta(branch(1), p(b)v q(b)) then
    [close(branch(1), 'd1.1', p(b)),
     close(branch(2), 'd1.2', q(b))]

Figure 4: Plan found by depth first and breadth first search