Implementing Tableaux

LeanTAP: Lean Tableau-based Deduction

The abstract of a 1995 paper in the Journal of Automated Reasoning:

```
prove((E,F),A,B,C,D) :- !, prove(E,[F|A],B,C,D).
prove((E,F),A,B,C,D) :- !, prove(E,A,B,C,D), prove(F,A,B,C,D).
prove(all(H,I),A,B,C,D) :- !,
  \+ length(C,D), copy_term((H,I,C),(G,F,C)),
  append(A,[all(H,I)],E), prove(F,E,B,[G|C],D).
prove(A,_,[C|D],_,_) :-
  ((A=-(B); -(A)=B)) -> (unify(B,C); prove(A,[],D,_,_)).
prove(A,[E|F],B,C,D) :- prove(E,F,[A|B],C,D).
```

implements a first-order theorem prover based on free-variable semantic tableaux. It is complete, sound, and efficient.


Implementing Tableaux

So how does this work then?

This lecture is largely based on the original paper by Beckert and Posegga (1995) and the article by Posegga and Schmitt (1999).

I’ve changed the programs a little bit though, in particular the preprocessing bits. The original is more compact and should be a little faster. My version should be easier to understand (... but please note that it has not been tested very carefully).


Representing Formulas

The authors of LeanTAP reuse existing Prolog operators (comma and semicolon) to represent formulas.

We'll use nice home-made operators instead:

\[- \text{op}(100, \text{fy}, \text{neg}), \]
\[- \text{op}(200, \text{yfx}, \text{and}), \]
\[- \text{op}(300, \text{yfx}, \text{or}), \]
\[- \text{op}(400, \text{yfx}, \text{implies}), \]
\[- \text{op}(500, \text{yfx}, \text{iff}). \]

Quantified formulas will be represented as follows:

- all(X,Fml)
- ex(X,Fml)

Variables will be represented using actual Prolog variables.

Example: all(X, all(Y, r(X,Y) implies r(Y,X) ) )

Negation Normal Form

The main program of LeanTAP assumes that the input is provided in negation normal form (NNF). That is, the only propositional connectives used are negation, conjunction, and disjunction, and negation only occurs right in front of atoms.

This makes the main program shorter, as we have to consider fewer tableau rules. Unlike for CNF, for instance, computing the NNF of a formula only takes linear time (so that’s ok).

For ease of presentation (it’s slightly less efficient), we split computing the NNF of a given formula into two subtasks:

(1) Eliminate any occurrences of implies and iff.
(2) Push negation inside for the resulting formulas.

So the overall program will have the following form:

\[- \text{nnf}(\text{Fml}, \text{NNF}) :- \text{eliminate}(\text{Fml}, \text{X}), \text{push}(\text{X}, \text{NNF}). \]

Prolog: Term Decomposition

Given a term T, the predicate =../2 (which is defined as an infix operator) can be used to generate a list, the head of which is the functor of T and the tail of which is the list of arguments of T:

\[- \text{?- loves(john,mary)} =.. \text{List}. \]
\[- \text{List} = [\text{loves, john, mary}] \]
\[- \text{Yes} \]

You can also use =../2 to compose new terms:

\[- \text{?- member(X, [f,g]), Y =.. [X,a,b].} \]
\[- X = f \]
\[- Y = f(a, b); \]
\[- X = g \]
\[- Y = g(a, b); \]
\[- \text{No} \]

Eliminating Non-NNF Operators

\[- \text{eliminate}(\text{neg A}, \text{neg Formula}) :- !, \text{eliminate}(A, \text{Formula}). \]
\[- \text{eliminate}(A \text{ implies B}, \text{Formula}) :- !, \text{eliminate}(\text{neg A or B}, \text{Formula}). \]
\[- \text{eliminate}(A \text{ iff B}, \text{Formula}) :- !, \text{eliminate}((A \text{ implies B}) \text{ and } \text{(B implies A)}, \text{Formula}). \]
\[- \text{eliminate}(\text{Formula}, \text{NewFormula}) :- \]
\[- \text{Formula} =.. [\text{Op},A,B], \text{member}([\text{Op}, [\text{and,or}]], !, \text{eliminate}(A, \text{NewA}), \text{eliminate}(B, \text{NewB}), \text{NewFormula} =.. [\text{Op},\text{NewA},\text{NewB}]]. \]
\[- \text{eliminate}(\text{Formula}, \text{NewFormula}) :- \]
\[- \text{Formula} =.. [\text{Op},X,A], \text{member}([\text{all,ex}]), !, \text{eliminate}(A, \text{NewA}), \text{NewFormula} =.. [\text{Op},\text{X,NewA}]. \]
\[- \text{eliminate}(\text{Atom}, \text{Atom}). \]
Pushing Negation Inside

push(neg neg A, NNF) :- !, push(A, NNF).
push(neg(A and B), NNF) :- !, push((neg A) or (neg B), NNF).
push(neg(A or B), NNF) :- !, push((neg A) and (neg B), NNF).
push(neg all(X,A), NNF) :- !, push(ex(X,neg A), NNF).
push(neg ex(X,A), NNF) :- !, push(all(X,neg A), NNF).
push(Formula, NNF) :-
  Formula =.. [Op,A,B], member(Op, [and,or]), !,
  push(A, NNF1), push(B, NNF2), NNF =.. [Op,NNF1,NNF2].
push(Formula, NNF) :-
  Formula =.. [Op,X,A], member(Op, [all,ex]), !,
  push(A, NNF1), NNF =.. [Op,X,NNF1].
push(Literal, Literal).

Skolemisation

The main program of LeanTAP does not implement a \textit{delta rule} (for existentially quantified formulas). So all existential quantification needs to be eliminated during preprocessing by means of \textit{Skolemisation}.

An outline of the Skolemisation algorithm (for formulas in NNF):

\begin{itemize}
  \item Step through the formula from the outside to the inside, and collect any universally quantified variables in a list \texttt{Vars}.
  \item Whenever you encounter an existentially quantified formula of the form \texttt{ex}(X,Fml):
    \begin{itemize}
      \item Generate a new Skolem function symbol \texttt{sk}$_{i}$.
      \item Replace any occurrence of \texttt{X} within \texttt{Fml} by \texttt{"sk}$_{i}$(\texttt{Vars})\texttt{"} to obtain \texttt{Fml}' and continue with \texttt{Fml}' in place of \texttt{ex}(X,Fml).
    \end{itemize}
\end{itemize}

Prolog: Assert and Retract

Prolog evaluates queries with respect to a knowledge base (your program + definitions of built-in predicates). It is possible to \textit{dynamically} add clauses to this knowledge base.

\begin{itemize}
  \item Executing a goal of the form \texttt{assert(+Clause)} will add the clause \texttt{Clause} to the Prolog knowledge base.
  \item Executing \texttt{retract(+Clause)} will remove that clause again.
  \item Using \texttt{retractall(+Clause)} will remove all the clauses matching \texttt{Clause}.
\end{itemize}

A typical application would be to dynamically create and manipulate a database. In that case the \texttt{Clauses} will usually be simple facts. Be careful when using \texttt{assert/1} and \texttt{retract/1}; they can make programs a lot more difficult to understand (and check).

Generating Skolem Function Symbols

Code to generate Skolem function symbols:

\begin{verbatim}
generate_new_symbol(Symbol) :-
  step_counter(Num), atom_concat(sk, Num, Symbol).
set_counter(Num) :-
  retractall(counter(_)), assert(counter(Num)).
step_counter(Num) :-
  counter(Num), Num1 is Num + 1, set_counter(Num1).
:- set_counter(1).
\end{verbatim}

If you run this, you will get a new symbol each time:

\begin{verbatim}
?- generate_new_symbol(S).  S = sk1
?- generate_new_symbol(S).  S = sk2
Yes
Yes
\end{verbatim}
Prolog: Copying Terms

Prolog comes with a built-in predicate `copy_term/2` that can be used to make a copy of a given term, whilst replacing all the variables in that term with new unbound variables.

It works as if it had been implemented like this:

```prolog
copy_term(Input, Output) :-
  assert(copy(Input)),
  retract(copy(Output)).
```

Example:

```prolog
?- copy_term(test(a,X,X), Term).
Term = test(a, _G254, _G254)
Yes
```

Substitution

We’ll have to be able to carry out a substitution `Var/Term` in a given expression. A very simple implementation uses `copy_term/2`. As `copy_term/2` will rename all variables by default, we need to state explicitly which ones cannot be renamed.

```prolog
substitute(Var/Term, Vars, Expression, Result) :-
  copy_term(Var:Vars:Expression, Term:Vars:Result).
```

Examples:

```prolog
?- substitute(X/f(Y), [], p(X), Formula).
Formula = p(f(Y))
Yes
?- substitute(X/f(Y), [Z], p(X,Z,U), Formula).
Formula = p(f(Y), Z, _G356)
Yes
```

Skolemisation

```prolog
skolem(NNF, SNNF) :- skolem(NNF, [], SNNF).
skolem(all(X,NNF), Vars, all(X,SNNF)) :- !,
  skolem(NNF, [X|Vars], SNNF).
skolem(ex(X,NNF), Vars, SNNF) :- !,
  get_new_symbol(F), SkolemTerm =.. [F|Vars],
  substitute(X/SkolemTerm, Vars, NNF, Fml).
  skolem(Fml, Vars, SNNF).
skolem(NNF1 and NNF2, Vars, SNNF1 and SNNF2) :- !,
  skolem(NNF1, Vars, SNNF1), skolem(NNF2, Vars, SNNF2).
skolem(NNF1 or NNF2, Vars, SNNF1 or SNNF2) :- !,
  skolem(NNF1, Vars, SNNF1), skolem(NNF2, Vars, SNNF2).
skolem(Literal, _, Literal).
```

Note: Variables need to be named apart for this to work correctly.

Examples

```prolog
?- skolem(ex(X,p(X)), Result).
Result = p(sk4)
Yes
?- skolem(all(X,all(Y,ex(Z,q(X,Y,Z)) or ex(U,p(U)))), Fml).
Fml = all(X, all(Y, q(X, Y, sk7(Y, X))or p(sk8(Y, X))))
Yes
?- nnf(all(X, p(X) implies ex(Y, r(X,Y))), NNF),
  skolem(NNF, SNNF).
NNF = all(X, neg p(X) or ex(Y, r(X, Y)))
SNNF = all(X, neg p(X) or r(X, sk12(X)))
Yes
```
LeanTAP: Overview

LeanTAP checks, for a given list of formulas in SNNF, whether the resulting tableau will close (i.e. whether the list is unsatisfiable).

Predicate: prove(+Fml, +UnExp, +Lits, +FreeV, +VarLim)
- Fml: the formula to which we want to apply a rule next
- UnExp: the rest of the current branch (unexpanded formulas)
- Lits: the literals encountered so far on the current branch
- FreeV: the list of free variables on the current branch
- VarLim: max. no. of free variables per branch (gamma rule)

Initialisation: Fml is the head of the input list; UnExp is the tail. Lits and FreeVars are []. VarLim is specified by the user.

The predicate succeeds when all branches can be closed.

LeanTAP: Alpha and Beta Rules

Because we assume the input to be in NNF, we have to consider only a single alpha and a single beta rule...

**Alpha rule**: proceed with the first conjunct and store the second in the list of unexpanded formulas.

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

**Beta rule**: first check that the left branch will close; then check the right branch.

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

Observe that variable instantiations made on one branch will carry over to the other (→ closure by unification).

LeanTAP: Gamma Rule

The gamma rule will only be applied if the list of free variables (FreeV) has not yet reached the maximum length (VarLim):

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).

The copy_term/2-line makes a copy of the matrix of the gamma formula, replacing X by X1 and making sure all the free variables stay intact (we could have used our substitute/4 instead).

The gamma formula is moved to the end of the list of unexpanded formulas (fairness); and we proceed with the new formula.

Prolog: Sound Unification

Prolog’s matching algorithm does not implement the occurs-check. This is a good thing, because Prolog would be very inefficient otherwise, but special care is required if in the rare cases where sound unification is important.

This is how SWI-Prolog reacts to a query where this matters:

?- X = f(X).
X = f(**)
Yes

It used to be something like this, which makes things clearer:

?- X = f(X).
X = f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f)))))))))))))))))))
Yes

But note that this is still wrong, if you want real unification...
Prolog: Sound Unification (cont.)

Fortunately, SWI-Prolog comes with a built-in predicate for sound unification that we can use. Examples:

?- unify_with_occurs_check(X, f(X)). No

?- unify_with_occurs_check(X, f(Y)). X = f(Y) Yes

An alternative would be to implement this ourselves, using Robinson’s algorithm ... but for now, we just abbreviate:

unify(X, Y) :- unify_with_occurs_check(X, Y).

LeanTAP: Literals

If none of the previous rules applied, then the formula in focus must be a literal. Succeed if its complement unifies with the head of the list of literals; otherwise recurse ...

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

Note the rather nifty use of {} to ensure that the head of the next rule below will never match this call to prove/5. That is, prove/5 is “abused” for checking unification for all members of Lits.

If the above fails (eventually), then we store the literal in focus in Lits and proceed with the next unexpanded formula:

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

One more time ...

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

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

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).

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

prove(Lit, [Next|UnExp], Lits, FreeV, VarLim) :-
  prove(Next, UnExp, [Lit|Lits], FreeV, VarLim).
Using LeanTAP

The following code makes using the program more convenient...

The predicate `prove/2` takes care of translating the list of input formulas into Skolem NNF and initialises `prove/5` correctly.

Recall that the second argument (VarLim) specifies the maximum number of applications of the gamma rule on each branch.

```prolog
prove(Input, VarLim) :-
    preprocess(Input, [Fml|Fmls]),
    prove(Fml, Fmls, [], [], VarLim).
```

Recursive application of the normalisation predicates:

```prolog
preprocess([], []).
preprocess([Fml|Fmls], [SNNF|SNNFs]) :-
    nnf(Fml, NNF), skolem(NNF, SNNF),
    preprocess(Fmls, SNNFs).
```

Examples

Answers are instant for examples such as these:

```prolog
?- prove([neg (all(X, (p or q(X))) iff (p or all(Y,q(Y)))]), 5).
Yes
?- prove([all(X, p(X) and q(X)), neg all(X,p(X))], 0).
No
?- prove([all(X, p(X) and q(X)), neg all(X,p(X))], 1).
Yes
?- prove([p or q, neg p or q, p or neg q, neg p or neg q], 0).
Yes
```

Advanced Prolog

In this short crash course, we have only discussed the very core features of the Prolog programming language. However, these should be sufficient for 95% of all tasks you are ever likely to face.

For the rest, your best source of information is usual the reference manual of your Prolog system.

A few “advanced features” (some of which were already mentioned today) are also discussed in the slides for my undergraduate class (“Lecture 6”, available from the usual place):

- Decomposing terms with 
- Collecting answers: `findall/3` etc.
- Dynamic predicates: `assert/1` and `retract/1`
- Input/output and file handling

Summary

- We have discussed the implementation of the LeanTAP prover in detail, including the translation into Skolem NNF.
- LeanTAP is a great example showing that Prolog allows you to write powerful programs in a simple and elegant manner.
- We have seen various new Prolog features along the way, namely 
  `=../2`, `assert/2` and `retract/2`, `copy_term/2`, `unify_with_occurs_check/2`, and `if-then-else` (`->`).