Introduction to Logic in Computer Science: Autumn 2006

Ulle Endriss
Institute for Logic, Language and Computation
University of Amsterdam

Plan for Today

Today we'll look into the interplay of logic and Prolog from two directions. First, we'll discuss the *logical foundations of Prolog:*

- Translating Prolog programs into Horn clauses
- Resolution as the reasoning engine underlying Prolog

Then we'll *apply Prolog to logic* and see how to implement a tableau-based theorem prover in Prolog. This also involves:

- Translation into negation normal form
- Skolemisation
- Sound unification (as opposed to Prolog's matching)

Along the way, we'll also see a few more Prolog features (`assert/1` and `retract/1`, `copy_term/2`, *if-then-else*, ...).

Horn Clauses

In logic, a *clause* is a disjunction of literals. A *propositional Horn clause* is a clause with at most one positive literal. Observe that:

$$\neg A_1 \vee \cdots \vee \neg A_n \vee B \quad \equiv \quad A_1 \wedge \cdots \wedge A_n \rightarrow B$$

A *first-order Horn clause* is a formula of the form $(\forall x_1) \cdots (\forall x_n) A$, with A being a propositional Horn clause.

"Pure" Prolog programs (without cuts, negation, or any built-ins with side effects) can be translated into sets of Horn clauses:

- Commas separating subgoals become $\wedge$.
- `:-` becomes $\rightarrow$, with the order of head and body switched.
- All variables are universally quantified (scope: full formula).
- Queries are translated as negated formulas ($Q \rightarrow \bot$).

Example

The following Prolog program (with a query) ...

```
bigger(elephant, horse).
bigger(horse, donkey).
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
?- is_bigger(elephant, X), is_bigger(X, donkey).
```

... corresponds to the following set of FOL formulas:

$$\{ \quad bigger(elephant, horse),$$

$$bigger(horse, donkey),$$

$$\forall x.\forall y.(bigger(x, y) \rightarrow is\_bigger(x, y)),$$

$$\forall x.\forall y.\forall z.(bigger(x, z) \wedge is\_bigger(z, y) \rightarrow is\_bigger(x, y))$$

$$\forall x.(is\_bigger(elephant, x) \wedge is\_bigger(x, donkey) \rightarrow \bot) \quad \}$$

Alternative notation: set of sets of literals (implicit quantification)

Prolog and Resolution

When Prolog resolves a query, it tries to build a proof for that query from the premises given by the program (or equivalently: it tries to refute the union of the program and the negated query).

Therefore, at least for pure Prolog, query resolution can be explained in terms of deduction in FOL. In principle, any calculus could be used, but historically Prolog is based on *resolution*.

What next?

- Resolution for full FOL
- Resolution for Horn clauses (to get a feel for why Prolog "works", despite the undecidability of FOL)

Binary Resolution with Factoring

Aim: Show $\Delta \models \varphi$ (for a set of sentences Δ and a sentence φ).

Preparation: Compute Skolem Normal Form of formulas in Δ and of $\neg\varphi$ and write them as a set of clauses (variables named apart).

Input: Set of clauses (which we want to show to be unsatisfiable).

Algorithm: Apply the following two rules. The proof succeeds if the empty clause (usually written as $\square$) can be derived.

Binary Resolution Rule
$$\frac{\{L_1\} \cup C_1 \quad \{L_2^c\} \cup C_2}{\mu(C_1 \cup C_2)}$$
L_2^c is the complement of L_2 μ is an mgu of L_1 and L_2

Factoring
$$\frac{\{L_1, \ldots, L_n\} \cup C}{\sigma(\{L_1\} \cup C)}$$
σ unifies $\{L_1, \ldots, L_n\}$

Why Factoring?

Try to derive the empty clause from the following (obviously unsatisfiable) set of clauses *without using the factoring rule*.

$$\{\, \{P(x), P(y)\},\ \{\neg P(u), \neg P(v)\} \,\}$$

$\Rightarrow$ It's not possible!

This means that our *binary* resolution rule alone (without factoring) would not be a *complete* deduction system for FOL.

Remark: The *general resolution rule* allows us to resolve using subclauses (rather than just literals). In that case we can do without factoring.

SLD Resolution for Horn Clauses

SLD Resolution stands for **S**elective **L**inear Resolution for **D**efinite clauses, where:

- *linear* means we always use the latest resolvent in the next step;
- we have a *selection function* telling us which literal to use; and
- the input is restricted to Horn clauses, all but one of which have to be *definite clauses* (that's another word for Horn clauses with *exactly* one positive literal).

SLD Resolution is complete for the Horn fragment (proof omitted).

SLD Resolution in Logic Programming

Prolog implements SLD Resolution:

- *Linearity:* we start with the only negative clause (the negated query) and then always use the previous resolvent (new query).
- The *selection function* is very simple: it always chooses the first literal (in the current "query").
- The input is restricted to *one negative Horn clause* (negated query) and a number of *positive Horn clauses* (rules and facts).

In practice, one problem remains: if there is more than one way to resolve with the selected literal (i.e. more than one matching rule or fact) then we don't know which one will eventually lead to a successful refutation. In Prolog, always the first one is chosen and if this turns out not to be successful, *backtracking* is used to try another one.

Worked Example

Consider the following Prolog program:

```
parent(elisabeth, charles).
parent(charles, harry).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).
```

What will happen if we submit the following query after the above program has been consulted by Prolog?

```
?- ancestor(elisabeth, harry).
```

Step 1: Translate into FOL

For the *program* we get the following formulas:

(1) $P(e, c)$

(2) $P(c, h)$

(3) $(\forall x)(\forall y)(P(x, y) \rightarrow A(x, y))$

(4) $(\forall x)(\forall y)(\forall z)(P(x, y) \wedge A(y, z) \rightarrow A(x, z))$

For the *negation of the query* we get:

(5) $\neg A(e, h)$

Step 2: Rewrite Formulas as Clauses

Formulas we get from translating a Prolog program already are in Prenex Normal Form and we don't need to Skolemise either (because there are no existential quantifiers).

We have to rewrite the implications as disjunctions. Here, we directly give the clauses (which correspond to disjunctions). Don't forget that variables have to be named apart.

(1) $\{P(e, c)\}$

(2) $\{P(c, h)\}$

(3) $\{\neg P(x_1, y_1),\ A(x_1, y_1)\}$

(4) $\{\neg P(x_2, y_2),\ \neg A(y_2, z_2),\ A(x_2, z_2)\}$

(5) $\{\neg A(e, h)\}$

Step 3: Apply SLD Resolution

(1)	$\{P(e,c)\}$	
(2)	$\{P(c,h)\}$	
(3)	$\{\neg P(x_1,y_1),\ A(x_1,y_1)\}$	
(4)	$\{\neg P(x_2,y_2),\ \neg A(y_2,z_2),\ A(x_2,z_2)\}$	
(5)	$\{\neg A(e,h)\}$	
(6)	$\{\neg P(e,y_3),\ \neg A(y_3,h)\}$	from (4,5) with $[x_2/e, z_2/h]$
(7)	$\{\neg A(c,h)\}$	from (1,6) with $[y_3/c]$
(8)	$\{\neg P(c,h)\}$	from (3,7) with $[x_1/c, y_1/h]$
(9)	$\square$	from (2,8)

Remark: If there had been variables in our query, then the substitutions made to them would have been part of the answer.

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.

B. Beckert and J. Posegga. *LeanTAP: Lean Tableau-based Deduction.* Journal of Automated Reasoning 15:339–358, 1995.

Implementing Tableaux

So how does this work then?

The remainder of today's lecture is largely based on the original paper by Beckert and Posegga (1995) and the review 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).

B. Beckert and J. Posegga. *LeanTAP: Lean Tableau-based Deduction.* Journal of Automated Reasoning 15:339–358, 1995.

J. Posegga and P.H. Schmitt. *Implementing Semantic Tableau.* Handbook of Tableau Methods, Kluwer, 1999.

Representing Formulas

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

We'll use nice home-made operators instead:

```
:- op(100, fy, neg),
   op(200, yfx, and),
   op(300, yfx, or),
   op(400, yfx, implies),
   op(500, yfx, 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 in 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:

```
nnf(Fml, NNF) :- eliminate(Fml, X), push(X, 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`:

```
?- loves(john,mary) =.. List.
List = [loves, john, mary]
Yes
```

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

```
?- member(X, [f,g]), Y =.. [X,a,b].
X = f
Y = f(a, b) ;
X = g
Y = g(a, b) ;
No
```

Eliminating Non-NNF Operators

```
eliminate(neg A, neg Formula) :- !, eliminate(A, Formula).

eliminate(A implies B, Formula) :- !,
  eliminate(neg A or B, Formula).

eliminate(A iff B, Formula) :- !,
  eliminate((A implies B) and (B implies A), Formula).

eliminate(Formula, NewFormula) :-
  Formula =.. [Op,A,B], member(Op, [and,or]), !,
  eliminate(A, NewA), eliminate(B, NewB),
  NewFormula =.. [Op,NewA,NewB].

eliminate(Formula, NewFormula) :-
  Formula =.. [Op,X,A], member(Op, [all,ex]), !,
  eliminate(A, NewA), NewFormula =.. [Op,X,NewA].

eliminate(Atom, 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 *delta rule* (for existentially quantified formulas). So all existential quantification needs to be eliminated during preprocessing by means of *Skolemisation*.

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

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

Prolog: Assert and Retract

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

- Executing a goal of the form `assert(+Clause)` will add the clause `+Clause` to the Prolog knowledge base.
- Executing `retract(+Clause)` will remove that clause again.
- Using `retractall(+Clause)` will remove *all* the clauses matching `Clause`.

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

Generating Skolem Function Symbols

Code to generate Skolem function symbols:

```
get_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).
```

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

```
?- get_new_symbol(S).
S = sk1
Yes
```

```
?- get_new_symbol(S).
S = sk2
Yes
```

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:

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

Example:

```
?- copy_term(test(a,X,X), Term).
X = _G181
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.

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

Examples:

```
?- substitute(X/f(Y), [], p(X), Formula).
X = _G182, Y = _G180, Formula = p(f(_G180))
Yes
?- substitute(X/f(Y), [Z], p(X,Z,U), Formula).
X = _G182, Y = _G180, Z = _G185, U = _G190,
Formula = p(f(_G180), _G185, _G341)
Yes
```

Skolemisation

```
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, 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).
```

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

Examples

For the following examples, I've simplified the output a little bit (if Prolog comes back with something like `X=_G182`, I'm just using `X`).

```
?- 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 ($\rightsquigarrow$ 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(...))))))))))
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(_G180)
Y = _G180
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).
```

Prolog: If-Then-Else

To satisfy a goal of the form `If -> Then ; Else`, Prolog will search for the *first* solution to the goal `If` and succeed if `Then` succeeds without backtracking into `If`. If `If` fails, then `Else` *must* succeed for the overall goal to succeed.

`If -> Then` is short for `If -> Then ; fail`.

If-then-else works as if (part of) `;/2` had been defined like this:

```
(If -> Then) ; _ :- If, !, Then.
(_  -> _) ; Else :- !, Else.
```

Without the else-part, it works as if implemented like this:

```
If -> Then :- If, !, Then.
```

This is not easy to get your head around, and I recommend to use this construct sparingly. In a nutshell, think of it as a "*local cut*".

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 initalises `prove/5` correctly.

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

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

Recursive application of the normalisation predicates:

```
preprocess([], []).

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

Examples

Answers are instant for examples such as these:

```
?- prove([neg all(X, (p or q(X))) iff (p or all(Y, q(Y)))], 5).
X = _G180, Y = _G190
Yes

?- prove([all(X, p(X) and q(X)), neg all(X,p(X))], 0).
No
```

I did not manage to get the following to work though (also not with the original LeanTAP)—but I may have overlooked a problem with the example and it's not actually a real challenge for LeanTAP ...

```
?- Ref = all(X, r(X,X)),
   Sym = all(X, all(Y, r(X,Y) implies r(Y,X))),
   Tra = all(X, all(Y, all(Z, r(X,Y) and (Y,Z) implies r(X,Z)))),
   Ser = all(X, ex(Y, r(X,Y))),
   Claim = Sym and Tra and Ser implies Ref,
   prove([neg(Claim)], 20).
```

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 `=../2`
- Collecting answers: `findall/3` etc.
- Dynamic predicates: `assert/1` and `retract/1`
- Input/output and file handling

Summary

- Pure Prolog corresponds to sets of Horn clauses.
- The reasoning engine underlying Prolog can be explained in terms of SLD Resolution.
- 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* (`->`).