Logisch Programmeren en Zoektechnieken

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

http://www.illc.uva.nl/~ulle/teaching/prolog/

Logic and Prolog

This final part of the course will be dedicated to the connections between logic and Prolog. This is a two-way street:

- Logical Foundations of Prolog:

  Prolog programs can be interpreted as sets of formulas in *first-order logic*. And when processing a query, Prolog is applying the rules of a logical proof system called *resolution*.

- Applications of Prolog to Logic:

  Prolog is a programming language that is particularly well suited to solving problems involving logic. We will see how to implement an *automated theorem prover* based on another proof system, *semantic tableaux*.

Correspondences

Prolog	**First-order Logic (FOL)**
predicate	predicate
argument	term
variable	universally quantified variable
atom	constant/function/predicate symbol
sequence of subgoals	conjunction
`:-`	implication (other way round)

Question

What is the logical meaning of this program?

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

Answer

The translation of a Prolog program is usually represented as a *set* of formulas, with each formula corresponding to one of the clauses in the program:

$$\{ \quad bigger(elephant, horse),$$
$$bigger(horse, donkey),$$
$$\forall x.\forall y.(bigger(x,y) \rightarrow is\_bigger(x,y)),$$
$$\forall x.\forall y.\forall z.(bigger(x,z) \wedge is\_bigger(z,y) \rightarrow is\_bigger(x,y)) \quad \}$$

Such a set is to be interpreted as the *conjunction* of all the formulas in the set.

Translation of Programs

- Predicates remain the same (syntactically).
- Commas separating subgoals become $\wedge$.
- `:-` becomes $\rightarrow$ and the order of head and body is changed.
- Every variable is bound by a universal quantifier ($\forall$), with the scope of the quantifier being the full implication.

Meaning of Prolog Programs

A Prolog program corresponds to a set of formulas, all of which are assumed to be true. This restricts the range of possible interpretations of the predicate and function symbols appearing in these formulas. The formulas in the translated program may be thought of as the premises in a proof.

If Prolog gives a positive answer to a given query, this means that the translation of the query is a logical consequence of these premises (possibly under the assumption of suitable variable instantiations). If Prolog gives a negative answer, this means that the query cannot be shown to be true under the assumption that all of the program formulas are true.

Translation of Queries

Queries are translated like rules; the "empty head" is translated as $\bot$ (falsum). This corresponds to the negation of the goal whose provability we are trying to test when submitting a query to Prolog:

$$Goal \rightarrow \bot \text{ is logically equivalent to } \neg Goal$$

Logically speaking, instead of deriving the goal itself, we try to prove that adding the negation of the goal to the program would lead to a contradiction:

$$Premises, (Goal \rightarrow \bot) \models \bot \quad \text{if and only if} \quad Premises \models Goal$$

Example

The query

```
?- is_bigger(elephant, X), is_bigger(X, donkey).
```

corresponds to the following first-order formula:

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

Horn Formulas

The (matrices of the) formulas we get when translating from Prolog all have the same structure:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \;\rightarrow\; B$$

Such a formula can be rewritten as follows:

$$\begin{aligned} A_1 \wedge A_2 \wedge \cdots \wedge A_n \;&\rightarrow\; B \quad &\equiv \\ \neg(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \;&\vee\; B \quad &\equiv \\ \neg A_1 \vee \neg A_2 \vee \cdots \vee \neg A_n &\vee B \end{aligned}$$

Hence, formulas obtained from translating Prolog formulas can always be rewritten as *disjunctions of literals* with at most one positive literal. Such formulas are know as *Horn formulas*.

Prolog and Resolution

The search tree built up by Prolog when trying to answer a query corresponds to a logic proof using *resolution*, which is a very efficient proof system for Horn formulas.

A short introduction can be found in the Lecture Notes; for more details refer to theoretically oriented books on logic programming.

Here we are only going to introduce resolution for propositional logic (corresponding to Prolog without variables) ...

Remark: In principle, any other proof system could be used as well (e.g., tableaux), but historically Prolog is based on resolution.

Propositional Resolution

Problem: The purpose of the resolution procedure is to show that a given set of formulas is *unsatisfiable*.

Why is that (proving unsatisfiability) an interesting problem?
Answer: $\mathcal{P} \cup \{\neg G\}$ is unsatisfiable *iff* G follows from the set $\mathcal{P}$.

Normal form: Recall that any formula can be transformed into an equivalent formula in *conjunctive normal form* (CNF). Also, a conjunction is satisfiable *iff* the set of its conjuncts is. Hence, it suffices to consider only *sets of disjunctions of literals* ...

Remark: In general, translation into CNF is not a trivial matter (e.g., your formula might get exponentially longer). But translated Prolog clauses already are in CNF, so we are fine here.

The Resolution Rule

Resolution for propositional logic consists of a single rule, which we apply to pairs of disjunctions until we find an explicit contradiction.

To simplify presentation, we write disjunctions as sets of literals.
Example: $A \vee \neg B$ becomes $\{A, \neg B\}$.

The resolution rule for propositional logic (and an example):

$$\frac{Dis_1 \cup \{L\} \qquad Dis_2 \cup \{\neg L\}}{Dis_1 \cup Dis_2} \qquad\qquad \frac{\{A,\ \neg B,\ C,\ D\} \qquad \{\neg C,\ D,\ \neg E\}}{\{A,\ \neg B,\ D,\ \neg E\}}$$

Make sure you understand why this is a *sound* rule ...

The simplification of $D \vee D$ to D is implicit (a.k.a. *factoring*).

An *explicit contradiction* is reached when both Dis_1 and Dis_2 are empty. In this case, we usually write $\square$ (or $\bot$) for the resolvent.

Examples

- Prove that P and $P \rightarrow Q$ entail Q (first translate!).
- Prove that the following set of formulas is unsatisfiable:

$$\{A \vee B,\ A \vee \neg B,\ \neg A \vee B,\ \neg A \vee \neg B\}$$

- Consider the following Prolog program:

```
happy :- summer, france.
happy :- rich.
rich :- lottery.
summer.
lottery.
```

  Simulate how Prolog would react to the following query by giving the appropriate resolution proof:

```
?- happy.
```

SLD Resolution

If all formulas are Horn formulas, then we can apply resolution in a "systematic" manner (called SLD resolution) and be sure that we always find a proof if there exists one at all. Intuitively, this is why Prolog "works" (reasoning in general FOL would be much harder).

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

- *Linearity:* we start with the only negative Horn formula (the 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 formula* (query) and a number of *positive Horn formulas* (rules and facts). Positive Horn formulas are also known as "definite clauses".

The above still leaves open which matching literal to choose; in practice this issue is dealt with via *backtracking*.

Matching vs. Sound Unification

Prolog's matching algorithm does not implement the so-called *occurs-check*. This is a good thing, because Prolog would be very inefficient otherwise, but special care is required 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(X)
Yes
```

It used to be something like this, which makes things much 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 ...

Sound Unification in SWI-Prolog

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
```

This predicate performs matching of the two arguments but also checks that the first does not occur (strictly) within the second.

Summary: Logic Foundations

- Prolog programs (without advanced features) correspond to sets of first-order logic (Horn) formulas.
- During translation, `:-` becomes an implication (from right to left), commas between subgoals become conjunctions, and all variables need to be universally quantified. Queries become (universally quantified) implications with $\bot$ in the consequent.
- Prolog's search to satisfy a query corresponds to a logical proof. In principle, any proof system could be used.

  Historically, Prolog is based on resolution, which is particularly suited as it is tailored to Horn formulas.
- For efficiency reasons, Prolog uses a simplified matching algorithm rather than full sound unification.

Automated Reasoning

We now want to implement a theorem prover in Prolog: a program that can check automatically whether a given set of formulas (the *premises*) logically entail a further formula (the *conclusion*).

We will restrict ourselves to *classical propositional logic* (but people have developed provers for many other logics as well).

Our prover will be based on the method of *semantic tableaux*.

Semantic Tableaux

Aim: Show that a given set of formulas is unsatisfiable (if you can do this, you can also do logical consequence or tautology checking).

Procedure: Start a *tableau* (a binary tree) with those formulas in the root. Apply *elimination rules* to formulas, resulting in new formulas being added to the end of a branch, and possibly in branches being split in two. *Close* a branch when it includes an obvious contradiction. Succeed if you can close all branches.

People use different ways of representing the formulas on the tableau. Some distinguish between formulas claimed to be true and those claimed to be false. You probably know this version:

formulas claimed to be true	*formulas claimed to be false*

For our implementation, it will be easier to explicitly *label* each formula as being either (claimed to be) *true* or *false*.

Elimination Rules and Branch Closure

Negation

$$\frac{t :: \neg A}{f :: A} \qquad \frac{f :: \neg A}{t :: A}$$

Conjunction

$$\frac{t :: A \wedge B}{\begin{array}{c} t :: A \\ t :: B \end{array}} \qquad \frac{f :: A \wedge B}{f :: A \mid f :: B}$$

Disjunction

$$\frac{t :: A \vee B}{t :: A \mid t :: B} \qquad \frac{f :: A \vee B}{\begin{array}{c} f :: A \\ f :: B \end{array}}$$

Closure

$$\frac{\begin{array}{c} t :: A \\ f :: A \end{array}}{\times}$$

Examples

Give tableau proofs for the following claims:

(1) $p \vee \neg p$ is a tautology

(2) $\{p \vee q,\, p \vee \neg q,\, \neg p\}$ is unsatisfiable

(3) $\neg p \vee \neg q \models \neg(p \wedge q)$ [logical consequence]

Semantic Tableaux in Prolog

In 1995, Bernhard Beckert and Joachim Posegga (Uni Karlsruhe) published a paper in the *Journal of Automated Reasoning* in which they showed how to implement a sound, complete, and efficient theorem prover for first-order logic based on semantic tableaux in Prolog requiring a total of 360 (!) characters (including spaces):

```
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(I,J),A,B,C,D) :- !,
   \+length(C,D),copy_term((I,J,C),(G,F,C)),
   append(A,[all(I,J)],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).
```

We will not try to understand this. But we will implement a prover for propositional logic using a very similar approach.

Representing Formulas

We need operators for the basic connectives of propositional logic:

```
:- op(100, fy, neg),
   op(200, yfx, and),
   op(300, yfx, or).
```

We also declare an operator that we will use to label formulas:

```
:- op(600, xfx, ::).
```

Examples:

$p \wedge \neg q$ claimed to be *true*	$\longrightarrow$	`t :: p and neg q`
$\neg(p \vee q)$ claimed to be *false*	$\longrightarrow$	`f :: neg (p or q)`

Representing Branches

A branch in a tableau is the list of formulas we find when we go from the root to one of the leaf nodes. We will model each branch in terms of *two* lists:

- `Forms`: the labelled non-atomic formulas (still) on the branch
- `Atoms`: the labelled atomic formulas on the branch

As long as there are still non-atomic formulas in `Forms`, we can continue to apply elimination rules. After we have applied a rule to a given formula, we remove that formula from `Forms`.

Eventually, only labelled atoms will be left (`Forms` will be empty).

Grand Plan

We will implement a predicate with this signature:

```
tableau(+Forms, +Atoms)
```

It should succeed if a tableau starting from a branch with the labelled complex formulas in `Forms` and the labelled atomic formulas in `Atoms` can eventually be closed (using our rules).

For example, to verify that $p \vee \neg p$ is a tautology we initialise the tableau with $p \vee \neg p$ (labelled as *false*) and an empty list of atoms:

```
?- tableau([f :: p or neg p], []).
Yes
```

That is, assuming that $p \vee \neg p$ is false allowed us to successfully close the tableau, i.e., to derive a contradiction.

Closing Branches

We can close a branch when we find the same formula on it twice, once labelled as *true* and once labelled as *false*.

We can restrict ourselves to closing branches using contradictorily labelled *atoms* only (think about this, it is not obvious).

- Closing only on atoms is a good strategy for a computer:

  Checking to see whether we can close a branch is the most costly operation (*why?*), so we don't want to do this too often.

- Closing on arbitrary formulas is a better strategy for humans:

  As a human, you want to build as small a tableau as you can.

Our rule for closing branches (base case of the recursion):

```
tableau(_, Atoms) :-          % don't need to know Forms
  member(t :: A, Atoms),
  member(f :: A, Atoms), !. % no need to backtrack
```

Elimination Rules: Negation

We translate each tableau elimination rule into a Prolog rule.
The idea is always the same:

(1) check whether the head of `Forms` is of a given type;

(2) apply the correct elimination rule to the head;

(3) continue proof search with the head being replaced by the new formula (this will be slightly more complex for branching rules).

Here are the two elimination rules for negation:

```
tableau([t :: neg A | Forms], Atoms) :- !,
  tableau([f :: A | Forms], Atoms).

tableau([f :: neg A | Forms], Atoms) :- !,
  tableau([t :: A | Forms], Atoms).
```

The list of `Atoms` doesn't change. The cut is applied once we know that this is the rule to use (after that we never want to backtrack).

Eliminating True Conjunctions

If the first formula on the branch is a conjunction (labelled *true*), then replace it with its two conjuncts (both labelled *true*):

```
tableau([t :: A and B | Forms], Atoms) :- !,
  tableau([t :: A, t :: B | Forms], Atoms).
```

Branching: Eliminating True Disjunctions

To close a branch with a (true) disjunctive formula on it, we need to split that branch and close the two resulting branches:

```
tableau([t :: A or B | Forms], Atoms) :- !,
  tableau([t :: A | Forms], Atoms),
  tableau([t :: B | Forms], Atoms).
```

Careful! We are *not* using disjunction (`;`) in the body of our rule. We need to show that the left branch will close eventually *and* that the right branch will close eventually.

Eliminating False Conjunctions and Disjunctions

These work exactly as the other rules we have seen:

```
tableau([f :: A and B | Forms], Atoms) :- !,
  tableau([f :: A | Forms], Atoms),
  tableau([f :: B | Forms], Atoms).

tableau([f :: A or B | Forms], Atoms) :- !,
  tableau([f :: A, f :: B | Forms], Atoms).
```

Final Step: Moving Atoms

We need one more rule. Whenever we encounter an atomic formula in `Forms` we need to move it to `Atoms`:

```
tableau([Label :: Atom | Forms], Atoms) :-
  atom(Atom), !,
  tableau(Forms, [Label :: Atom | Atoms]).
```

That's it! And it works:

```
?- tableau([t :: p, t :: neg p or q, f :: q], []).
Yes

?- tableau([t :: p or neg p], []).
No
```

Note that the order of the rules of the program does not matter: any order will lead to a correct program. But the order might affect efficiency. Can you predict what would be a good order?

User Interface: Tautology Checking

The following predicate takes a formula and then appropriately initialises a tableau to check whether it is a tautology:

```
tautology(Formula) :-
  tableau([f :: Formula], []).
```

Examples:

```
?- tautology(p or neg p).
Yes
```

```
?- tautology(neg (neg (p or (q and r)) and p) or q).
Yes
```

Summary: Automated Reasoning

This has been a case study in automated reasoning:

- We have seen how to implement a tableau-based theorem prover in Prolog. The main data structure is that of lists of formulas (representing branches) and elimination rules are implemented as operations on the heads of those lists.
- Despite its simplicity, our program will perform rather well also on larger examples.
- Note that our implementation only checks whether a closed tableau exists for a given set of formulas—it does not return that tableau to the user! Of course, you can also write a program for that task, but it is a bit more difficult.

Automated reasoning is an important topic in AI and people have developed all sorts of automated reasoners for different types of problems, modelled in different logics as well as other formalisms.