CS2LAP: Logic and Prolog 2000/2001
Implementing a Goal-directed Theorem Prover in Prolog

Abstract. This paper describes how a simple automated theorem prover for classical propositional logic, which is based on the goal-directed deduction system, can be implemented in Prolog. A partially written program is available from the course web site and it is your task to complete this program. If you answer the questions one by one, you will end up with a complete theorem prover, that, for example, is capable of solving all the goal-directed problems for propositional logic from the textbook, coursework sheets, and exam papers.

1 Introduction

A partially completed theorem prover based on the goal-directed method for propositional logic can be downloaded from this web site (follow the link to *Labs*):

`http://www.dcs.kcl.ac.uk/teaching/units/cs2lap/`

The program consists of the following four files: `main.pl`, `prover.pl`, `ready.pl`, and `auxiliary.pl`.

Question 1 *Download the four program files and save them together in one directory. Now start Prolog and compile the program by submitting the following query:*

```
?- consult( main).
```

This will also compile the other three program files. Look at `main.pl` *to understand why.*

Given this program, you can already prove *some* logical arguments, but some of the goal-directed rules are not yet implemented. Also the translation into ready for computation form is not yet fully implemented.

Question 2 *Use the Prolog prover to generate a goal-directed proof for the following argument (which happens to be in ready for computation form already):*

$$\vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$$

To do this you have to submit the following query to the interpreter:[1]

```
?- prove( [], ((a implies b) implies a) implies a, []).
```

Study Prolog's reply to this query and try to understand the proof.

The rest of this paper is structured as follows: Section 2 briefly reviews the goal-directed theorem proving method and discusses a few refinements that we have to make to allow for a simple implementation. Section 3 provides an overview of the contents of the four program files that make up the prover. After that, Section 4 guides you through the completion of the actual prover itself (file `prover.pl`) and Section 5 does the same for the translation into ready for computation form (file `ready.pl`).

[1] The first empty list in the query denotes the empty set of data formulas for this example. The other empty list is the initially empty history of goals. This is where candidate formulas for the restart rule are stored – more on this later.

 Version: November 21, 2000

2 The Goal-directed Calculus Reviewed

Given a set of data formulas in ready for computation form[2] and a goal formula (which has to be a conjunction of formulas in ready for computation form) we can use the following goal-directed rules to show that the goal logically follows from the data:[3]

1. If the current goal A is atomic and can be found in the data, **succeed**.
2. If $\perp$ can be found in the data, **succeed**.
3. **Rule for implication:** to show $\mathcal{P} \vdash A \Rightarrow B$ show $\mathcal{P}, A \vdash B$.
4. **Rule for conjunction:** to show $\mathcal{P} \vdash A \wedge B$ show $\mathcal{P} \vdash A$ and $\mathcal{P} \vdash B$.
5. **Rule for atoms:** to show $\mathcal{P}, A \Rightarrow B \vdash B$ show $\mathcal{P}, A \Rightarrow B \vdash A$.
6. **Rule for atoms (with $\perp$):** to show $\mathcal{P}, A \Rightarrow \perp \vdash B$ (B atomic) show $\mathcal{P}, A \Rightarrow \perp \vdash A$.
7. **Restart rule:** at any time, we can replace our current goal with a previous one.

During lectures you have heard already about the problem of *looping*: if an argument is not valid, i.e. if there is no proof, we might fall into an infinite loop trying to apply the same rules again and again. For the case of classical propositional logic this problem can easily be solved when we make the following observations:

1. We should use the restart rule only if no other rules can be applied any more.
2. We should use restart only with atoms. When restarting other goal formulas (implications or conjunctions) the following step would just be a repetition of what has been done earlier on already.
3. We should only use restart, if directly after that we can apply a rule different from restart (otherwise the first application of restart would have had no effect at all). This means, when considering to restart a previous goal B (an atom) we have to check whether either B or $A \Rightarrow B$ (for some A) can be found in the data. In the former case we would succeed in the next step, in the latter case we could apply the rule for atoms in the next step. There are no other cases.
4. When applying one of the rules for atoms, the implication we used can be deleted from the data.

Given these refinements we get a sound and complete deduction system which always terminates.

In a Prolog implementation, formulas that we want to be able to reuse with the restart rule have to be stored in a list. In the sequel we shall refer to this list as the *history* of previous goals.

[2] For details on the translation into ready for computation form see Section 5.

[3] For more details see *Elementary Logics*, Chapter 4, in particular pages 93–95 and 103–106.

3 Program Overview

Our Prolog program consists of four program files: `main.pl`, `prover.pl`, `ready.pl`, and `auxiliary.pl`. The latter implements some auxiliary predicates, `ready.pl` provides predicates for translating formulas into ready for computation form, and `prover.pl` implements the goal-directed rules. The logic operators are defined in `main.pl` and this file also contains queries to compile the other three program files. Both `prover.pl` and `ready.pl` are not complete yet.

The following list provides an overview of all the predicates implemented:

`prove/2` (file `main.pl`): The main predicate: it uses `ready/2` and `ready_list/2` to translate formulas into ready for computation form and then initiates the actual proof using the predicate `prove/3`.

`prove/3` (file `prover.pl`): This is the implementation of the goal-directed rules (see Section 4).

`output/4` (file `prover.pl`): This predicate is used by `prove/3` to print each proof step on the screen.

`ready/2` (file `ready.pl`): Translate a given formula into a conjunction of formulas in ready for computation form (see Section 5).

`ready_list/2` (file `ready.pl`): Translate a list of formulas into a list of formulas in ready for computation form (see Section 5).

`add/3` (file `auxiliary.pl`): Add an element to a list, if it is not already a member of that list. Used to add goals to the history (list of candidates for restart) for the implementation of `prove/3`.

`conj2list/2` (file `auxiliary.pl`): Convert a conjunction into the list of its conjuncts. Used in `prove/3` for the rule for implication (in case the antecedent is a conjunction it has to be split up) and for the implementation of `ready_list/2`.

Question 3 *Browse through the four files to get a first idea of how the program works. Note that you can test some of the predicates independently from the others. Try for example the following queries:*

```
?- conj2list( neg a and (a or b) and c, Result).
?- output( [c, d implies false], a implies b, [], 'rule name').
```

4 Implementing the Theorem Prover

The core of the entire program is the implementation of the predicate `prove/3` (file `prover.pl`). The first argument is a list of data formulas and the second argument position holds the current goal formula. Here we assume that all formulas are in ready for computation form already. In the third argument position the history of previous goals is stored. That means, whenever the current goal changes and the last goal has been an atom, this atom is added to the history list so it can later be retrieved from there again when applying the restart rule.

Each of the Prolog rules for `prove/3` corresponds to one of the goal-directed rules given in Section 2. If none of these seven rules applies the predicate should fail, because no proof is possible.[4]

For example, the first rule for success is implemented as follows:

```
prove( Data, A, History) :-
  atom( A),
  member( A, Data), !,
  output( Data, A, History, 'success').
```

We test that the current goal `A` is an atom and can be found in the list of data formulas `Data`. Then we call `output/4` to print out this information. In this rule, as well as in all others, we insert a cut at the point where it is sure that this is the correct rule to use, which means no alternative rules should be tried in case of a later failure.

Another example is the rule for atoms:

```
prove( Data, B, History) :-
  atom( B),
  member( A implies B, Data), !,
  output( Data, B, History, 'rule for atoms'),
  delete( Data, A implies B, NewData),
  add( B, History, NewHistory),
  prove( NewData, A, NewHistory).
```

This rule should be used, if the current goal `B` is atomic and a formula `A implies B` can be found in `Data`. The new list of data formulas is obtained by deleting that implication from the list `Data`. `B` has to be added to the history. Then `prove/3` is called again with the new goal formula `A`.

Question 4 *Study the implementation of the predicate* `prove/3` *from* `prover.pl` *and fill in the missing Prolog clauses for the second case of success ($\perp$ in data),*[5] *the rule for conjunction, and the rule for atoms with $\perp$. Test your implementation.*

5 Translation into Ready for Computation Form

A formula is in ready for computation form, if it is either an atom (or $\perp$) or if it is an implication, where the consequent is an atom (or $\perp$) and the antecedent is a conjunction of formulas in ready for computation form.

[4] Because we don't just want Prolog to fail in this case, but to put out a message to the user beforehand, we have implemented an eighth rule for calling `output/4` before `prove/3` fails.

[5] In the implementation, use the Prolog atom `false` to represent $\perp$.

Any propositional formula can be translated into a conjunction of formulas in ready for computation form by exploiting the following equivalences:

$$
\begin{array}{lcl}
A \Rightarrow (B \wedge C) & \equiv & (A \Rightarrow B) \wedge (A \Rightarrow C) \\
A \Rightarrow (B \Rightarrow C) & \equiv & (A \wedge B) \Rightarrow C \\
\neg A & \equiv & A \Rightarrow \bot \\
A \vee B & \equiv & (A \Rightarrow \bot) \Rightarrow B
\end{array}
$$

Because the corresponding translation rules would have to be applied to every matching subformula, such an approach is not particularly feasible for a straightforward implementation (despite being perfectly reasonable for humans). Therefore, we define a bigger set of rules in such a way that it is sufficient to apply them systematically from the outside to the inside.

A formula A is recursively rewritten as A^* using the following steps:[6]

$$
\begin{array}{lcl}
(A \Rightarrow (B \wedge C))^* & = & (A \Rightarrow B)^* \wedge (A \Rightarrow C)^* \\
(A \Rightarrow (B \Rightarrow C))^* & = & (A \wedge B \Rightarrow C)^* \\
(A \Rightarrow (B \vee C))^* & = & ((A \wedge (B \Rightarrow \bot)) \Rightarrow C)^* \\
(A \Rightarrow \neg B)^* & = & ((A \wedge B) \Rightarrow \bot)^* \\
(A \Rightarrow q)^* & = & A^* \Rightarrow q, \quad q \text{ atomic} \\
(A \Rightarrow \bot)^* & = & A^* \Rightarrow \bot \\
(\neg A)^* & = & (A \Rightarrow \bot)^* \\
(A \vee B)^* & = & ((A \Rightarrow \bot) \Rightarrow B)^* \\
(A \wedge B)^* & = & A^* \wedge B^*
\end{array}
$$

A formula to which none of these rules applies is already in ready for computation form (base case). For every propositional formula A the result A^* of the translation process will be a conjunction of formulas in ready for computation form, which is logically equivalent to A.

Such a rule system can be translated directly into a Prolog program. This has (partly) been done in `ready.pl`. The predicate `ready/2` corresponds to the translation function $()^*$. Although not all rules have been implemented yet, you can already use it to translate *some* formulas into ready for computation form.

Question 5 *Translate the formula $\neg(A \wedge \neg A)$ into ready for computation form. To do this submit the following query to Prolog:*

```
?- ready( neg (a and neg a), Result).
```

The first translation rule, for example, has been implemented in Prolog as follows:

```
ready( A implies (B and C), X and Y) :- !,
  ready( A implies B, X),
  ready( A implies C, Y).
```

[6] See also *Elementary Logics*, page 104; but note that the third and the fourth rule are missing there.

Whenever the first argument (the input formula) matches the pattern `A implies (B and C)` this rule will be used. Because of the cut, backtracking will not be possible (we know that if this rule applies no other rule should be tried instead). In the rule body `ready/2` is used recursively twice: `A implies B` is translated and the result is matched with the variable `X`, and the same for `A implies C` and the variable `Y`. The overall result is then the term `X and Y`.

Question 6 *Study the implementation of the predicate* `ready/2` *from* `ready.pl`*. Implement the missing Prolog rules for the following translation steps:*

$$\begin{array}{lcl} (A \Rightarrow (B \vee C))^* & = & ((A \wedge (B \Rightarrow \bot)) \Rightarrow C)^* \\ (A \Rightarrow \neg B)^* & = & ((A \wedge B) \Rightarrow \bot)^* \\ (A \vee B)^* & = & ((A \Rightarrow \bot) \Rightarrow B)^* \end{array}$$

Test your implementation with some examples.

The predicate `ready_list/2` applies `ready/2` to all members of a list of formulas. It also breaks up conjunctions into parts and inserts each of the parts into the resulting list. This predicate is used in `prove/2` (see `main.pl`) to translate the list of data formulas into a list of formulas in ready for computation form.

6 Conclusion

If you have answered all questions correctly you should now have a fully functional theorem prover for classical propositional logic. You can use `prove/2` from `main.pl` instead of `prove/3` from `prover.pl`. This predicate will do the translation into ready for computation form for you and it will also initialise the history of goals with the empty list (the third argument in `prove/3`).

Question 7 *Use your Prolog prover to generate goal-directed proofs for the following arguments:*

1. $\vdash\ ((A \Rightarrow B) \Rightarrow A) \Rightarrow (A \vee B)$
2. $A \Rightarrow C, B \Rightarrow D\ \vdash\ A \wedge B \Rightarrow C \wedge D$
3. $\vdash\ \neg(A \wedge \neg A)$

The first one, for example, can be produced by typing in the following query:

```
?- prove( [], ((a implies b) implies a) implies (a or b)).
```