Logisch Programmeren en Zoektechnieken

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

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

Backtracking, Cuts and Negation

In this lecture, we are going to look in more detail into how Prolog evaluates queries, in particular into the process of backtracking.

We are going to discuss both the use of backtracking and some associated problems, and introduce a way of explicitly controlling backtracking (via so-called *cuts*). We are also going to discuss the closely related subject of negation.

Backtracking

Choicepoints: Subgoals that can be satisfied in more than one way provide *choicepoints*. Example:

```
..., member(X, [a, b, c]), ...
```

This is a choicepoint, because the variable `X` could be matched with either `a`, `b`, or `c`.

Backtracking: During goal execution Prolog keeps track of choicepoints. If a particular path turns out to be a failure, it jumps back to the most recent choicepoint and tries the next alternative. This process is known as *backtracking*.

Clever Use of Backtracking

Given a list in the first argument position, the predicate `permutation/2` generates all possible permutations of that list in the second argument through enforced backtracking (if the user presses `;` after every solution):

```
permutation([], []).

permutation(List, [Element | Permutation]) :-
  select(Element, List, Rest),
  permutation(Rest, Permutation).
```

Recall that `select/3` checks whether the element in the first argument position can be matched with an element of the list in the second argument position; if so, the term in the third argument position is matched with the remainder of that list.

Example

```
?- permutation([1, 2, 3], X).
X = [1, 2, 3] ;
X = [1, 3, 2] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
No
```

Problems with Backtracking

Asking for alternative solutions generates wrong answers for this predicate definition:

```
remove_duplicates([], []).

remove_duplicates([Head | Tail], Result) :-
  member(Head, Tail),
  remove_duplicates(Tail, Result).

remove_duplicates([Head | Tail], [Head | Result]) :-
  remove_duplicates(Tail, Result).
```

Example

Example:

```
?- remove_duplicates([a, b, b, c, a], List).

List = [b, c, a] ;

List = [b, b, c, a] ;

List = [a, b, c, a] ;

List = [a, b, b, c, a] ;

No
```

Introducing Cuts

Sometimes we want to prevent Prolog from backtracking into certain choicepoints, either because the remaining alternative choices would yield wrong solutions (like in the previous example) or for efficiency reasons.

This is possible by using a *cut*, written as `!`. This built-in predicate always succeeds and prevents Prolog from backtracking into subgoals placed *before* the cut inside the same rule body.

Example

The correct program for removing duplicates from a list:

```
remove_duplicates([], []).

remove_duplicates([Head | Tail], Result) :-
  member(Head, Tail), !,
  remove_duplicates(Tail, Result).

remove_duplicates([Head | Tail], [Head | Result]) :-
  remove_duplicates(Tail, Result).
```

Cuts

Parent goal: When executing the subgoals in a rule's body the term *parent goal* refers to the goal that caused the matching of the head of the current rule.

> Whenever a cut is encountered in a rule's body, all choices made between the time that rule's head has been matched with the parent goal and the time the cut is passed are final, i.e., any choicepoints are being discarded.

Exercise

Using cuts (but without using negation), implement a predicate `add/3` to insert an element into a list, if that element isn't already a member of the list. Make sure there are no wrong alternative solutions. Examples:

```
?- add(elephant, [dog, donkey, rabbit], List).
List = [elephant, dog, donkey, rabbit] ;
No

?- add(donkey, [dog, donkey, rabbit], List).
List = [dog, donkey, rabbit] ;
No
```

Solution

```
add(Element, List, List) :-
  member(Element, List), !.

add(Element, List, [Element | List]).
```

Problems with Cuts

The predicate `add/3` does not work as expected when the last argument is already instantiated! Example:

```
?- add(dog, [dog, cat, bird], [dog, dog, cat, bird]).
Yes
```

We could use the following implementation of `add/3` instead:

```
add(Element, List, Result) :-
  member(Element, List), !,
  Result = List.

add(Element, List, [Element | List]).
```

While this solves the problem, it also emphasises that using cuts can be tricky and affects the declarative character of Prolog ...

Summary: Backtracking and Cuts

- *Backtracking* allows Prolog to find all alternative solutions to a given query.
- So: Prolog provides the search strategy, not the programmer! This is why Prolog is called a *declarative* language.
- Carefully placed *cuts* (`!`) can be used to prevent Prolog from backtracking into certain subgoals. This may make a program more efficient and/or avoid the generation of (wrong) alternative answers.
- On the downside, cuts can destroy the declarative character of a Prolog program (which, for instance, makes finding mistakes a lot more difficult).

Prolog's Answers

Consider the following Prolog program:

```
animal(elephant).
animal(donkey).
animal(tiger).
```

... and the system's reaction to the following queries:

```
?- animal(donkey).
Yes

?- animal(duckbill).
No
```

The Closed World Assumption

In Prolog, `Yes` means a statement is *provably true*. Consequently, `No` means a statement is *not provably true*. This only means that such a statement is *false*, if we assume that all relevant information is present in the respective Prolog program.

For the semantics of Prolog programs we usually do make this assumption. It is called the *Closed World Assumption*: we assume that nothing outside the world described by a particular Prolog program exists (is true).

The \+-Operator

If we are not interested whether a certain goal succeeds, but rather whether it fails, we can use the \+-operator (negation). `\+ Goal` succeeds, if `Goal` fails (and *vice versa*). Example:

```
?- \+ member(17, [1, 2, 3, 4, 5]).
Yes
```

This is known as *negation as failure:* Prolog's negation is defined as the failure to provide a proof.

Negation as Failure: Example

Consider the following program:

```
married(peter, lucy).
married(paul, mary).
married(bob, juliet).
married(harry, geraldine).

single(Person) :-
  \+ married(Person, _),
  \+ married(_, Person).
```

Example (continued)

After compilation, Prolog reacts as follows (recall that Mary is married to Paul, while our little database didn't mention Claudia):

```
?- single(mary).
No

?- single(claudia).
Yes
```

In the closed world described by our Prolog program Claudia has to be single, because she is not known to be married.

Where to use \+

Note that the \+-operator can only be used to negate *goals*. These are either (sub)goals in the *body of a rule* or (sub)goals of a *query*.

We cannot negate facts or the heads of rules, because this would actually constitute a redefinition of the \+-operator (in other words: an explicit definition of Prolog's negation, which wouldn't be compatible with the closed world assumption).

Disjunction

We already know *conjunction* (comma) and *negation* (`\+`). We also know *disjunction*, because several rules with the same head correspond to a disjunction.

Disjunction can also be implemented directly within one rule by using `;` (semicolon). Example:

```
parent(X, Y) :- father(X, Y); mother(X, Y).
```

This is equivalent to the following program:

```
parent(X, Y) :- father(X, Y).

parent(X, Y) :- mother(X, Y).
```

Example

Write a Prolog program to evaluate a row of a truth table. (Assume appropriate operator definitions have been made beforehand.)

Examples:

```
?- true and false.
No


?- true and (true and false implies true) and neg false.
Yes
```

You may want to use the `call/1` predicate to invoke a goal associated with a given variable. Example:

```
?- X = write('Hello world!'), call(X).
Hello world!
X = write('Hello world!')
Yes
```

Solution

```
% Falsity
false :- fail.   % not needed for SWI-Prolog

% Conjunction
and(A, B) :- call(A), call(B).

% Disjunction
or(A, B) :- call(A); call(B).
```

Solution (continued)

```
% Negation
neg(A) :- \+ call(A).

% Implication
implies(A, B) :- call(A), !, call(B).
implies(_, _).
```

Note

We know that in classical logic, $\neg A$ is equivalent to $A \rightarrow \bot$.

Similarly, instead of using `\+` in Prolog, we could define our own negation operator as follows:

```
neg(A) :- call(A), !, fail.
neg(_).
```

Summary: Negation and Disjunction

- *Closed World Assumption:* In Prolog everything that cannot be proven from the given facts and rules is considered false.
- *Negation as Failure:* Prolog's negation is implemented as the failure to provide a proof for a statement.
- Goals can be negated using the `\+`-operator.

  Always use `\+`, not the `not`-operator, as the latter may mean different things in different Prolog systems.
- A disjunction of goals can be written using `;` (semicolon). (The comma between two subgoals denotes a conjunction.)