

COMP210: Artificial Intelligence

Lecture 12. Prolog: Some advanced features

Trevor Bench-Capon
Room 215, Ashton Building

<http://www.csc.liv.ac.uk/~tbc/COMP210>

Overview

- In this lecture we discuss further aspects of Prolog in particular:-
 - preventing backtracking (cut);
 - negation as failure (using the cut fail combination);
 - the closed world assumption.

Search and Backtracking

- Prolog does depth-first search
- If a goal fails, Prolog will automatically *backtrack* and explore another possibility
 - Useful programming concept, and often very useful
 - BUT: Uncontrolled backtracking may lead to inefficiency (or worse)

Example: Tax Rates

- Consider the following program that sets tax rates for particular incomes.


```
tax_rate(X,0):- X < 10000.
tax_rate(X,1):- 10000 =< X, X < 30000.
tax_rate(X,2):- 30000 =< X.
```
- Note, we assume that X is instantiated to a number, as required by the comparison operators.
- The goal `tax_rate(2000,Z)` will succeed with `Z=0`.

Backtracking

- Now consider the query
 - `?- tax_rate(2000,Z), 1 < Z.`
- When executing `tax_rate(2000,Z)` Z is instantiated as 0.
- The second goal becomes `1 < 0` which fails.
- At this point Prolog will try to backtrack and match the first goal with each of the other definitions for `tax_rate` which will also fail.

Tracing this program

```
% trace
| ?- tax_rate(2000,Z), 1<Z.
  1      1 Call: tax_rate(2000,_449) ?
  2      2 Call: 2000<10000 ?
  2      2 Exit: 2000<10000 ?
  1      1 Exit: tax_rate(2000,0) ?
  1      1 Redo: tax_rate(2000,0) ?
  3      2 Call: 10000=<2000 ?
  3      2 Fail: 10000=<2000 ?
  4      2 Call: 30000=<2000 ?
  4      2 Fail: 30000=<2000 ?
  1      1 Fail: tax_rate(2000,_449) ?
no
```

Annotations in the original image:

- `Z = 0` points to the first `Exit` line.
- `Could stop here` points to the `no` line.
- `2nd clause fails` points to the `Fail` line for the second clause.
- `3rd clause fails` points to the `Fail` line for the third clause.

Problem

- The problem is that the set of rules for `tax_rate` are *mutually exclusive*, i.e. for any value of `X` the body of only one rule can succeed.
- So we know that once `1 < 0` fails there is no point in trying any other clause for `tax_rate`.
- We do not want to backtrack once a solution is found

Cut

- We can tell Prolog explicitly not to backtrack by using **cut**, written as `!`.
- The program with cuts becomes the following.


```
tax_rate(X,0):- X < 10000, !.
tax_rate(X,1):- 10000 =< X, X < 30000, !.
tax_rate(X,2):- 30000 =< X.
```
- Now with the query


```
?- tax_rate(2000,Z), 1 < Z.
```
- the program does not backtrack to explore the branches from rules 2 and 3, i.e. the program is more efficient.
- With the query


```
* ?- tax_rate(40000,Z), 1 < Z.
```
- The program does not reach the cut in the first clause and so does backtrack to try the second and the third clauses and so finds the solution `Z = 2`

Green Cuts and Red Cuts

- With this type of cut the program gives the same results as the version without cuts but is, in general, more efficient. Such cuts are called *green cuts*.
- This is not always the case though. Using cuts can also change the results of the program. These are known as *red cuts*. Examples of red cuts include omitting explicit conditions in a rule.

Red Cuts—Tax program

- Now we remove the condition `X < 10000` in the second rule, the idea being that if we get to try the second rule then `X < 10000` must have already failed.


```
- tax_rate(X,0):- X < 10000, !.
- tax_rate(X,1):- X < 30000, !.
- tax_rate(X,2):- 30000 =< X.
```
- Note that because of the cut the query


```
?- tax_rate(500,Z).
```

 will only produce one result, i.e. `Z=0`.
- What would happen if we posed the same query to the above program without cuts?

Example

- Here is a program computing the maximum of two numbers.


```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```
- The following attempt to use cuts is wrong.


```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) .
```
- This is wrong as the query `max(5,1,1)` will succeed.
- A correct, if unnatural, solution is to change the first rule to the following.


```
- max(X,Y,Max) :- X >= Y, !, X = Max.
```

Negation as Failure

- `fail` is a built in goal that always fails.
- `true` is a built in goal that always succeeds.
- `cut` and `fail` are often used in combination:


```
different(X,X):- !,fail.
different(_,_).
```
- If the two inputs to 'different' match, the first clause is matched.
- The cut means the program is committed to this choice and cannot subsequently match with the second 'different' clause.
- The goal 'fail' results in backtracking but no further choices can be made due to the cut.

Negation as Failure

```
| ?- different(a,b).
yes
| ?- different(a,a).
no
```

- If two inputs that do not match are input to different they cannot be matched with the first clause.
- They can be matched with the second clause which succeeds.
- In summary different(X,Y) means if X and Y match then different(X,Y) fails, otherwise different(X,Y) succeeds.

Not Member

- The cut fail combination is useful for "not" queries eg `notmember`.

```
member(H, [H|Tail]).
member(X, [_|Tail]):-
    member(X, Tail).
```

```
notmember(X, List):- member(X, List), !, fail.
notmember(_, _).
```

- Most versions of Prolog have a built in operator **not** where **not Goal** is true if **Goal** is not true.
 - `Notmember(X, List):- not member(X, List)`
- Be careful using both cut and the cut fail combination as they sometimes produce counter intuitive results.

Family Tree Again

- Consider our original family tree program

```
female(cathy). % Cathy is female.
female(lucy).
female(pauline).
female(lou).
```

- and the following queries.

```
| ?- female(cathy).
yes
| ?- female(peggy).
no
```

- This means there is not possible to show that peggy is female. Peggy could be male, or it could be that we do not know

Closed World Assumption

- The world is closed in the sense that everything that exists is stated in the program or can be derived from the program.
- If something is not in the program (or cannot be derived from it) then it is not true and consequently its negation is true.
- The program has all the facts:
 - All students have a record on Spider
- The rules provide a complete procedure
 - No other way of showing the required fact

Closed World Assumption

- If we added

```
notfemale(X):- female(X),!,fail.
notfemale(_).
```

- and tried the following queries:-

```
| ?- notfemale(cathy).
no
| ?- notfemale(peggy).
yes
```

- The latter is counter intuitive, since we don't expect to have every female in our database.
- It should not be understood as *Peggy is not female*, But as *there is no information in the program to prove peggy is female*.

Closed World Assumption

- The Closed World Assumption can sometimes be valid:

```
suit(clubs).
suit(diamonds).
suit(hearts).
suit(spades).
```

We have all the suits, so we can accept a no to `suit(wands)` as really false.

CWA for Rules

```
student(X):- undergraduate(X).
student(X):- postgraduate(X).
```

There is no other way for a person to be a student so a no to student(trevor) means that trevor is not a parent (spider records all UGs and PGs)

We (implicitly) complete the database with

```
not student(X) if not (postgraduate(X) or
undergraduate(X))
```

i.e – student(X) if and only if undergraduate(X) or postgraduate(X)

If this completion is acceptable we can make the CWA.

Summary

- We have looked at the use of cut to improve the efficiency of programs by stopping backtracking.
- Green cuts prune branches (affect the procedural behaviour of the program) without changing the results of the program.
- Red cuts both prune branches and affect the results.
- Cut makes it possible to introduce *negation as failure* via the cut fail combination.
- Cuts may destroy the correspondence between the declarative and procedural semantics.
 - Negation of failure corresponds to “real” negation only of the closed world assumption is valid. Otherwise it only means cannot be shown.
- Use cuts with care. Only use cut with good reason.