Tutorium to Introduction to AI, 9th week - Nicolas Höning

Nicolas Höning

July 13, 2006

the cut - operator

- What was that again?
- Why would anyone use that?
- green vs red cut

an example

- Occasionally, backtracking and multiple answers are annoying. Prolog provides the cut symbol (!) to control backtracking. The following code defines a predicate where the third argument is the maximum of the first two.
- max(A,B,M) :- A < B, M = B.
  max(A,B,M) :- A >= B, M = A.
- The code may be simplified by describing the case in the signature (the head of the predicate). You know that already.
  max(A,B,B) :- A < B.
  max(A,B,A) :- A >= B.

an example

- This is a nice example, because it shows, once more, that often Prolog programming is about dividing the search space. Here we divide it into A $<$ B, and A $>=$ B.
- But in this case, we are kind of repeating ourselves. I want to define one half of the search space, not both:
  max(A,B,B) :- A $<$ B.
  max(A,B,A).
- However, because Prolog backtracks all possible solutions, now incorrect answers can result as is shown here:
  ?- max(3,4,M).
  M = 4;
  M = 3

an example

- To prevent backtracking to the second rule the cut symbol is inserted into the first rule:
  max(A,B,B) :- A < B,!.
  max(A,B,A).
- The cut says: "If you prove this predicate until here, this branch of the search tree is all that you should try out (on this level of the tree)." Don't backtrack other solutions for this level.
- In our case, the erroneous answer will not be generated. We used cut as a kind of "else" statement here.

Why would anyone use that?

- **efficiency reasons** - the other part of the tree gets cut, there will be no checks on predicates that might be expensive.
- cuts are also great when you can't describe your case any better than saying: "If not the other case, then do this". **It's a kind of general "else"**
- **cut can be a better not()**
  in our example, we could describe both parts of the search space in a positive manner:
  A < B;A >= B.
  But sometimes the one part of the search space is the provable part, and the other one is the "unprovable" part. As Prolog's not() means just that anyway (not(X): X is not provable), a cut is an elegant solution here.

green cuts

- green cuts are just for efficiency reasons, for example:
  do_homework(X) :- wants_to(X),!.
  do_homework(X) :- needs_points(X), not(wants_to(X)).
- We don't need to check that second predicate when the first succeeded. But we did not take away any code.

red cuts

- red cuts make the program shorter, for example:
  do_homework(X) :- wants_to(X),!.
  do_homework(X) :- needs_points(X).
  That code does the same, and it's shorter (and mark the elegant absence of not()).
  But this time we depend on the cut operator! Without it, the code is not the same anymore!
- You need to do that right. The chances of messing the code up rise.
- For example, you should not forget what the cut was intended to do!

red cuts

The difficulties that can lead to errors are:

- the cut is not made in the part of the tree that is actually cut
- one cut can cut zero to infinitely many branches, who knows?
- the order problem: Another Prolog implementation might execute the rules in random order, or another programmer (or you - 5 days later) doesn't notice the importance of the ordering (so you should at least comment your code well).
- I'm sure I don't know them all, there may be more