Logik für Informatiker: PROLOG
Part 6: Cuts & Negation

Andreas Karwath

&

Wolfram Burgard

(original slides by Peter Flach)
Cut
Definition:
“! succeeds immediately and commits Prolog to all the choices since the parent goal was unified with the head of the clause the cut occurs in.“ (Sterling/Shapiro: The Art of Prolog, p.158)

\[
\begin{align*}
\text{p}(X) & : \text{-} \text{q}(X, Y), !, \text{r}(X), \text{s}(Y, Z). \\
\text{p}(X) & : \text{-} \text{q}(Y, X).
\end{align*}
\]

All clauses of a predicate following a clause with a cut in its body are not considered in a proof, once the cut succeeds (is proven)

Alternative solutions for the subgoals preceding the cut cannot be found anymore.

The cut does not influence the subgoals within the same clause after the cut.
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(john,paul).
mother(mary,paul).

?-parent(john,C)
:-father(john,C), !
:-mother(john,C)
[]

parent(X,Y) :- father(X,Y), !.
parent(X,Y) :- mother(X,Y).
father(john,paul).
mother(mary,paul).

Pruning by means of cut
The effect of cut

\[\text{p}(X,Y) :\neg \text{q}(X,Y).
\text{p}(X,Y) :\neg \text{r}(X,Y).
\text{q}(X,Y) :\neg \text{s}(X), !, \text{t}(Y).
\text{r}(c,d).
\text{s}(a).
\text{s}(b).
\text{t}(a).
\text{t}(b).\]
parent(X,Y):-father(X,Y),!.
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).

?-parent(john,C)
:-father(john,C),!.
:-mother(john,C)

?-parent(P,paul)
:-father(P,paul),!.
:-mother(P,paul)
• Reduces the search space by dynamic pruning of the search tree

• Can be used to prevent the Prolog system from searching parts of the search space that are known to contain no solutions.

• **Green cuts** do not change the meaning of the program, **red cuts** do (to be avoided!)
Example for reasonable use of green cuts: Definition of relation \( \text{polynomial}(\text{Term, } X) \) for recognizing whether a term is a polynomial in \( x \).

Useful for distinguishing cases that exclude each other.
polynomial(X,X) :- !.
polynomial(Term,X) :- constant(Term),!.
polynomial(Term1+Term2,X) :-
    !,polynomial(Term1,X),
    polynomial(Term2,X).
polynomial(Term1-Term2,X) :-
    !,polynomial(Term1,X),
    polynomial(Term2,X).
polynomial(Term1*Term2,X) :-
    !,polynomial(Term1,X),
    polynomial(Term2,X).
polynomial(Term1/Term2,X) :-
    !,polynomial(Term1,X),
    constant(Term2).
polynomial(Term^N,X) :-
    !,natural_number(N),
    polynomial(Term,X).
Red cuts: explicitly omitting conditions = changing the meaning of a program

Thus, problematic; also most common source of bugs in Prolog programs

An (almost) reasonable use: definition of relation

```
if_then_else(P,Q,R)
```
Negation as Failure
Negation as failure

- Consider:
  
  ```prolog
  p :- q, !, r.
p :- s.
  ```

  means roughly:
  
  ```prolog
  p :- q, r.
p :- not_q, s.
  ```

  Define `not_q`:
  
  ```prolog
  not_q :- q, !, fail.
  not_q.
  ```

- Not very practical. Better:
  
  ```prolog
  not(Goal) :- call(Goal), !, fail.
  not(Goal).
  ```

- `not/1, call/1` are meta-predicates.
Draw an SLD tree for the query:

?- likes(A, B)
given the following program:

likes(peter, Y) :- friendly(Y).
likes(T, S) :- student_of(S, T).
student_of(maria, peter).
student_of(paul, peter).
friendly(maria).

Add a cut, in order to „remove“ one of the answers
A->peter, B->maria

and show the result in the SLD tree. Can this be done without removing the third answer?
Exercise

likes(peter,Y):-friendly(Y).
likes(T,S):-student_of(S,T).
student_of(maria,peter).
student_of(paul,peter).
friendly(maria).

likes(peter,Y):-!,friendly(Y).  likes(T,S):-student_of(S,T),!.

?-likes(A,B)

:-friendly(B), :-student_of(B,A)

[[]
A=peter
B=maria
[[]
A=peter
B=paul

?:-likes(A,B)

:-friendly(B), :-student_of(B,A)

[[]
A=peter
B=maria
[[]
A=peter
B=paul

?-likes(A,B)

:-friendly(B), :-student_of(B,A), !

[[]
A=peter
B=maria
[[]
A=peter
B=paul

?-likes(A,B)

:-friendly(B), :-student_of(B,A), !

[[]
A=peter
B=maria
[[]
A=peter
B=paul
\[\begin{align*}
p & : - q, r. \\
p & : - \text{not}(q), s. \\
& s.
\end{align*}\]

\[\begin{align*}
\text{not}(\text{Goal}) & : - \text{Goal}, !, \text{fail}. \\
\text{not}(\text{Goal}).
\end{align*}\]
\[-\text{not}(q)\]
\[\text{p} :- \text{not}(q), r.\]
\[\text{p} :- q.\]
\[\text{q}.\]
\[\text{r}.\]

\[-\text{not}(\text{Goal}) :- \text{Goal}, !, \text{fail}.\]
\[-\text{not}(\text{Goal}).\]
Prolog’s `not` is unsound
Red cuts: explicitly omitting conditions = changing the meaning of a program

Thus, problematic; also most common source of bugs in Prolog programs

An (almost) reasonable use: definition of relation

\texttt{if\_then\_else}(P,Q,R)
Declarative meaning: relation is true if $P$ and $Q$ are true, or if not $P$ and $Q$ are true.
Operational meaning: if $P$ is shown, then $Q$, else $R$.

% if_then_else(P,Q,R) :-
% either $P$ and $Q$ or not $P$ and $R$
if_then_else(P,Q,R) :- call(P),!,call(Q).
if_then_else(P,Q,R) :- call(R).

Using the cut, it is implicitly known in the second clause, that $P$ has failed in the first clause.
Useful programming device: if-then-else

- Syntax: *(if-condition -> then-branch; else-branch)*
- Only the first solution of *if-condition* is explored.

```prolog
max(Xs, X) :- max(Xs, -inf, X).

max([], Max, Max).
max([X|Xs], MaxSoFar, Max) :-
    (X > MaxSoFar ->
    max(Xs, X, Max)
    ;
    max(Xs, MaxSoFar, Max)
    ).
```
Show that \( \text{?- } p \) succeeds, but that \( q \) und \( r \) are used twice.

Show that \( q \) and \( r \) are used exactly once.
Exercise

p :- q, r, s, !, t.
p :- q, r, u.
q.
r.
u.

p :- q, r, if_s_then_t_else_u.
if_s_then_t_else_u :- s, !, t.
if_s_then_t_else_u :- u.
q.
r.
u.

?- p

: - q, r, if_s_then_t_else_u
     
     : - r, if_s_then_t_else_u
         
     : - r, !, t
         
     : - s, !, t
         
     : - u
         []
     : - u
         []

: - q, r, if_s_then_t_else_u
     
     : - r, if_s_then_t_else_u
         
     : - r, !, t
         
     : - s, !, t
         
     : - u
         []