5.4 The Cut Predicate and Negation

Montag, 26. Juni 2017 09:00

Cut Predicate allows to control the backtracking mechanism of Prolog. With the cut, one can also implement negation.

male(X) :- ¬female(X).

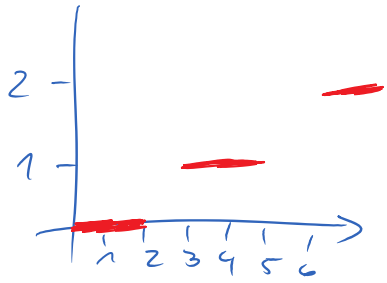
is not allowed in pure LP, because

{male(X), female(X)}

is not a Horn clause.

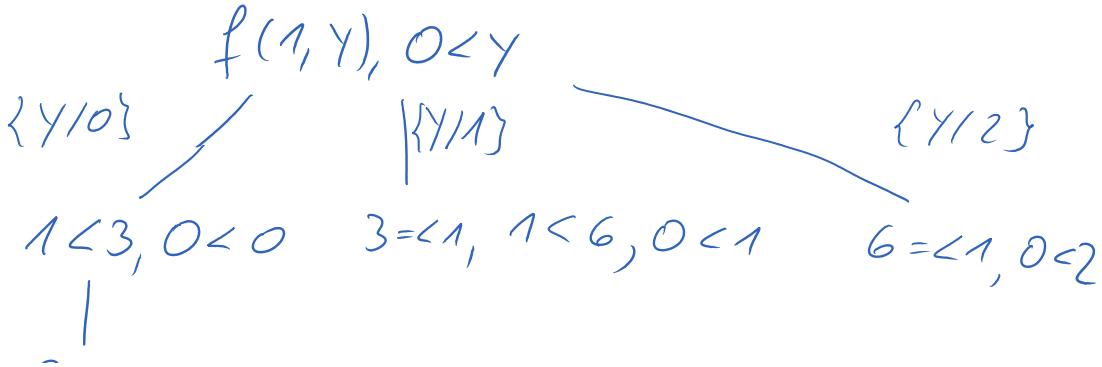
5.4.1. The Cut Predicate

Ex: $f(X) = \begin{cases} 0, & \text{if } X < 3 \\ 1, & \text{if } 3 \leq X < 6 \\ 2, & \text{if } 6 \leq X \end{cases}$



Slide 30

```
f(X, 0) :- X < 3.
f(X, 1) :- 3 =< X, X < 6.
f(X, 2) :- 6 =< X.
Query: ?- f(1, Y), 0 < Y.
```



0<0

Finite Failure, Answer: false

Can we make this program more efficient?

Observation ↑: The conditions of the 3 prog. clauses exclude each other. If one of these conditions is true, then we should not consider the other prog. clauses anymore, because their conditions will be false.

This can be exploited by adding the cut predicate to our clauses.

Cut: ! predicate of arity 0

Proof of ! always succeeds, but as a side effects it prunes alternative alternative paths in the SLD tree.

```
f(X,0) :- X<3, !.
f(X,1) :- 3=<X, X<6, !.
f(X,2) :- 6=<X.
```

Effect: if proof of X<3 succeeds, then no alternative proofs for X<3 and f(X,0) are considered

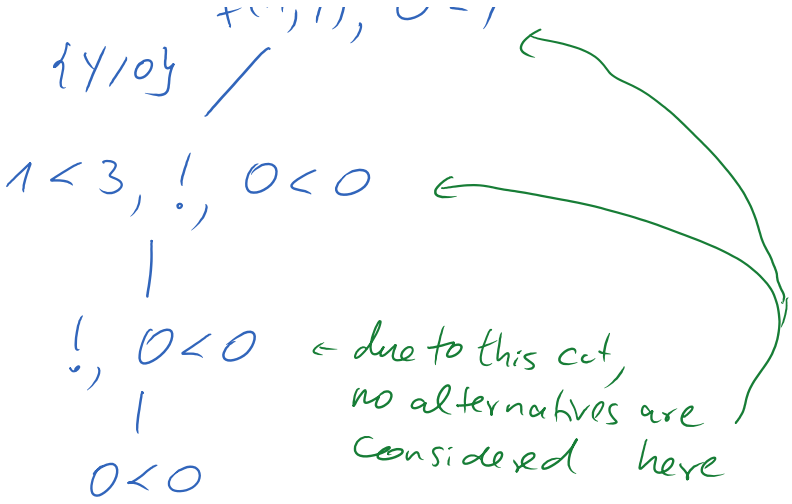
↳ disregard the remaining 2 f-clauses

Slide30

↑ Similar effect

f(1,Y), 0<Y

{Y/0}

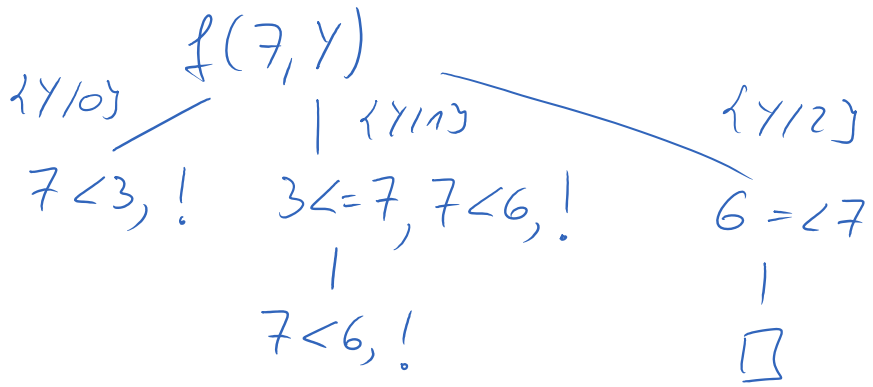


Green Cuts: influence efficiency + termination, but not the results of the program.
If one removes the cuts, one would still get the same solutions.

One can improve efficiency even further:

?- f(7, Y).

Slide 31



Answer: Y = 2

Observation 2: X < 3 and 3 =< X are complementary. If X < 3 fails, then we do not have to check 3 =< X

anymore, because it must be true.
(Similarly for $X < 6$ and $6 =< X$).

```
f(X,0) :- X<3, !.
f(X,1) :- X<6, !.
f(X, 2).
```

Slide 31

Red Cuts: If one removes the cuts, then one obtains different results.

Then ?- f(1, Y).

Y = 0 ; Y = 1 ; Y = 2

?- f(gerd, 2)
true

?- f(0, 2)
true

When adding cuts, one typically has a certain form of query in mind where certain arguments are input (ground terms) and others are output (variables). Here: first arg of f should be input, second arg of f should be output.

Exact Meaning of Cuts

Slide 32

If a query ?- $A_1, \ldots, A_k$ is resolved with a

prog. clause $B :- C_1, \ldots, C_i, !, C_{i+1}, \ldots, C_n$
(using the mgu τ of A_1 and B) then one first has to solve the instantiated literals $C_1, \ldots, C_i$ before reaching !.
Effect of the cut: When backtracking, no further alternatives for $C_1, \ldots, C_i$, $B \hat{=} A_1$ are explored.

Ex: To explain the full effect of the cut. Slide 33

```
?- a(X).
X=0 ; X=1 ; X=2 ; X=3 ; X=4 ; X=5
```

If one adds a cut in the second b-clause, then one obtains: Slide 34

```
?- a(X).
X=0 ; X=1 ; X=5
```

Examples to demonstrate the use of cuts: Slide 35

```
gcd(X, 0, X) :- !.
gcd(0, X, X) :- !.
gcd(X,Y,Z) :- X =< Y, !, Y1 is Y-X, gcd(X, Y1, Z).
gcd(X,Y,Z) :- X1 is X-Y, gcd(X1, Y, Z).
```

(gcd is only intended for use with nat. numbers)

Another exp.:

remove(X, Xs, Ys) should hold if the list Ys results from the list Xs by removing all occurrences of X.

```
?- remove(1, [0,1,2,1], Ys).
Ys = [0,2].
```

```
remove(_, [], []).
remove(X, [X|Xs], Ys) :- !, remove(X, Xs, Ys).
remove(X, [Y|Xs], [Y|Ys]) :- remove(X, Xs, Ys).
```

The cut ensures that Clause 3 is only reached if the first element of the list is different from X. Otherwise:

```
?- remove(1, [0,1,2,1], Ys).
```

This would have the answers

Ys = [0,2]; Ys = [0,2,1]; Ys = [0,1,2]; Ys = [0,1,2,1]

5.4.2. Meta-Variables and Negation

up to now: Variables are instantiated by terms

now: Variables can also be instantiated by formulas (meta-variables)

Ex. `p(a).` ← p is a pred. sym. of arity 1 but its arg. is a formula, not a term

`q.` ← q is a predicate symbol of arity 0

Since Prolog is untyped, it does not distinguish between pred. symbols and fct. symbols (i.e., atomic formulas and terms).

```
?- p(X), X.
X = a
?- p(X), X, Y.
error
```

← not allowed. Meta-Variables have to be instantiated before they are used in resolution

With meta-variables, we can implement Boolean connectives.

Disjunction:

```
or(X, Y) :- X.
or(X, Y) :- Y.
```

← A similar predicate is pre-defined as the infix operator ; with the directive `:-op(1100, xfy, ;).`

```
?- X=4 ; X=5.
X = 4 ← 
```

```
?- or(X=4, X=5).
X = 4 ; X = 5
```

X=4 ; X=5 | X=4 ; X=5

If-Then-Else: if A then B else C

if(A, B, C) :- A, !, B.
if(A, B, C) :- C.

Cut is needed to ensure that if the proof of A succeeds then we don't read the 2nd clause.

A similar predicate is pre-defined in Prolog:

$A \rightarrow B ; C$

With the cut, we can also implement negation. Up to now, we can only derive statements of the form $\exists X_1, \ldots, X_n \; A_1 \wedge \ldots \wedge A_k$

($A_1, \ldots, A_k$: atomic formulas)

Now we also want to include negated atomic formulas.

For negation, Prolog uses 2 assumptions:

(a) Closed World Assumption: all true statements can be derived from the prog. clauses. Thus: If a statement can't be derived,

then it must be false.

(5) If a statement can't be derived from the program, then this can be detected in finite time.

Then: negation can be implemented as "finite failure":
To prove $\neg A$, one tries to prove A. If this proof fails in finite time (i.e., if the SLD tree for A is finite and does not contain $\square$), then $\neg A$ holds.

Thus, one can implement "not" as follows:

```
not(A) :- A, !, fail.
not(A).
```

← pre-defined pred. which always fails

← needed to ensure that one doesn't read the second clause if A could be proved

"not" is pre-defined, can also be used as a prefix operator `\+`

```
not_equal(X,Y) :- not(X=Y).
```

```
?- not_equal(1,2).
true

?- X=2, not_equal(1,X).
```

```
?- not_equal(1,1).
false

?- not_equal(1,X), X=2.
```

```
?- X=2, not_equal(1,X).
X=2
```

```
?- not_equal(1,X), X=2.
false

?- not_equal(1,X).
false
```

```
not_equal(1,X)
      |
  not (1 = X)
      |
  1=X, !, fail
      | {X=1}
    !, fail
      |
    fail
```

$1=X$ true if $\exists X.\ 1=X$ holds

$1=X$ false if $\forall X.\ 1 \neq X$ holds

not_equal(1,X) true if — " —

Behavior of negation when assumptions ⓐ or ⓑ do not hold:

Slide 36

```
even(0).
even(X) :- X1 is X-2, even(X1).
```

From these clauses, one can't derive even(1).

```
?- not (even(1)).
non-termination
```

```
?- even(1)
non-termination
```

Problem: Assumption ⑤ does not hold.

```
even(0).
even(X) :- X >= 2, X1 is X-2, even(X1).
```

```
?- not(even(-2)).
true
```

Programmer forgot to specify that -2 is even. CWA assumes that this was on purpose, i.e., that even(-2) is false.

Problem: Ass. ④ doesn't hold.

Alternative correct version:

```
even(0) :- !.
even(X) :- X > 0, !, X1 is X-1, not(even(X1)).
even(X) :- X1 is X+1, not(even(X1)).
```