5.4 The Cut Predicate and Negation

Montag, 15. Juni 2015 08:30

5.4.1: Built-in Cut-Predicate

Goal: do not traverse certain parts of the SLD-tree when backtracking
(in order to increase efficiency or to avoid non-termination)

5.4.2: implement Meta-Predicates like negation

$$\text{female}(X) :- \text{not}(\text{male}(X)).$$

↑ SLD a negation was not available yet

Reason: $\{ \text{female}(X), \text{male}(X) \}$ is no Horn Clause

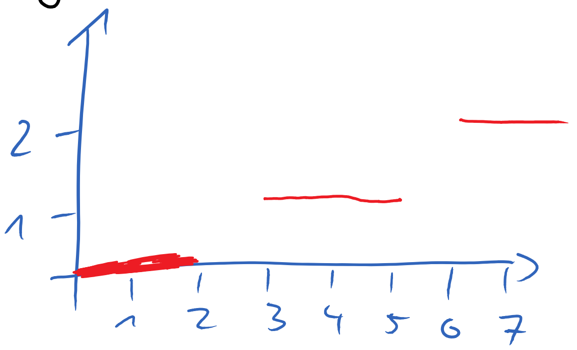
5.4.1 The Cut Predicate

Backtracking: if one reaches finite failure or if user enters ";" after reaching □.

Cut: avoid certain backtracking

Ex:

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



f(X 0) :- X<3.

| ?- f(1,Y) 0<Y

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

```
?- f(1,Y), 0<Y.
```

Observation: The conditions

$X<3$

$3=<X$, $X<6$

$6=<X$

exclude each other.

If proving one of these conditions succeeds, one should not backtrack to try the other f-clauses.

Solution: Cut predicate !

- Predicate of arity 0
- Proof of ! always succeeds
- Side effect: as soon as ! has been proved, certain alternative paths of the SLD tree are not explored anymore.

The cuts in this program are "green cuts": only influence efficiency, but if one removes the cuts, one still gets the same results.

Efficiency of example program can be improved further: $?- f(7, Y).$

If $X<3$ succeeds in the first clause, the we will not read clause 2+3 (because of !).

If $X<3$ fails in the first clause, then there is no need to check $3 =< X$ in clause 2, because $\neg X<3$ implies $3 =< X$.

$\Rightarrow$ remove $3=<X$ from clause 2
remove $6=<X$ from clause 3.

Now the cuts are "red cuts". Removing the cuts would yield different new answer substitutions: $?- f(0,2).$ ← if cuts are removed

true

In general: What is the effect of a cut?

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 mgu σ of A_1 and B,

then one obtains the SLD-tree on the slide.

Cut means that no alternatives are considered anymore for those nodes between

$$A_1, \dots, A_k \quad \text{and}$$

$$\sigma'(\,!, C_{i+1}, \dots, C_n, A_2, \dots, A_k).$$

But for the nodes above and below those two nodes, backtracking works as usual.

Example to illustrate the full effect of cut:

Version without cuts.

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

Now replace the second b-clause by

```
b(X) :- c(Y), !, d(X,Y).
?- a(X).
X=0 ; X=1 ; X=5
```

Examples for using the cut in natural programs:

- gcd (greatest common divisor)

```
?- gcd(12,9,Z).
Z=3
```

- remove

remove(X, Xs, Ys) iff Ys results from Xs by removing all occurrences of the element X from the list Xs.

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

The cut in clause 2 is needed to ensure that clause 3 is only reached if $X \neq Y$ (i.e., if the element to be removed is not at the beginning of the list).

If this cut were deleted, we would get:

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

5.4.2. Meta-Variables and Negation

Prolog allows the use of meta-variables:

Variables : can be instantiated by terms

meta-Variables: — " ——— formulas

(terms: monika, date(15, 6, 2015), ...

formulas: male(gerd), married(gerd, renate), ...)

Prolog also allows meta-predicates:

predicate: applied to terms

meta-predicate: applied to formulas

Ex:

p(a). ← p is a meta-predicate (applied to formula a)

a. ← a is a predicate symbol

?- p(X), X. ← X is a meta-variable

X=a

?- X. ← no resolution with completely uninstantiated meta-variables

program error

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

is pre-defined under the name ";"
using the directive

```
:- op(1100, xfy, ;)
```

Thus, one can ask query:

```
?- X=4 ; X=5.
```

```
X=4 ; X=5
```

One can also implement a meta-predicate for "if-then-else":

if(A,B,C) should implement "if A then B else C"

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

Cut is needed to ensure that one does not reach clause 2 if A holds

if(A,B,C) is pre-defined in Prolog under the name "A → B ; C"

<u>Negation</u> is implemented as "finite failure" ("Negation as failure"):

("Negation as failure"):

Goal: prove $\neg A$

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

← pre-defined predicate whose proof always fails

is pre-defined in Prolog, can also be used as prefix-operator `\+`

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

?- not_equal(1,2).
true

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

← Negation turns $\exists$ into $\forall$.

More precisely: one has to prove not(1=X).
To this end, prove 1=X, succeeds, thus not(1=X) fails.

Negation in Prolog uses two assumptions:

1. If a query doesn't hold, then this is determined in finite time.

But:
```
?- not(even(1)).
```

does not return "true", because even(1) doesn't terminate. ↝ Although "even(1)" doesn't hold, we can't detect it in finite time.

2. Closed World Assumption: If something can't be proved with our program, then it must be false.

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

Since proof of even(-2) fails.