Laboration 1 - Logik för dataloger DD1351

Emil Ståhl

Uppgift 1

Först binds a till X,

Sedan kommer systemet försöka att binda Y till X, vilket kommer att misslyckas eftersom X redan är bunden till a.

Därefter kommer Y att bindas till b.

Detta resulterar dock i att systemet kommer ge error eftersom den andra bindningen ej kan genomföras.

Uppgift 2

Negation as failure innebär att för att negera p kommer systemet att försöka bevisa p / exekvera p. Om exekveringen av p lyckas, kommer dess negation att misslyckas. Likväl som att om p misslyckas leder det till att negationen lyckas.

Exempel på negation as failure

Följande predikat definieras i en databas.

```
president(obama).
ej_president(X) :- \+  president(X).
```

Följande fråga ställs till systemet:

```
?- ej_president(mccain).
```

Systemet kommer då att substituera `X` mot `mccain` i predikatet `ej_president(X)` och därefter försöka bevisa `president(mccain).` Då detta inte är definierat i vår databas misslyckas exekveringen av `president(mccain)` vilket leder till att predikatet `ej_president(mccain)` lyckas.

Ännu en fråga ställs till systemet:

```
?- ej_president(obama).
```

Denna fråga kommer då att misslyckas eftersom att det finns ett definierat predikat i databasen som motsvarar `president(obama).`

Man kan därför se det som att det som står till vänster om ”/+” alltid kommer vara motsatsen till det som står till höger.

Uppgift 3

Vi vill definiera ett predikat som givet en lista som representerar en sekvens skapar en annan lista där sista elementet är borttaget. Elementet skall returneras i det tredje argumentet. Till exempel:

?- findlast([1,2,3], R,E).

skall generera R = [1,2] och E = 3.

Lösning:

```
findlast([H],[], H).
findlast([H|T],[H|Rest], E):-
  findlast(T, Rest, E).
```

Följande output ges:

```
E = 3
R = [1,2].
```

Beskrivning av exekvering:

Först anropas predikatet `findlast([1,2,3], R, E).`

Det prolog gör då är att reservera minnesadresser för R respektive E som följer

```
Call: findlast([1,2,3],_285,_286).
```

`findlast` anropas rekursivt med ”svansen”.

```
Call: findlast([2,3],_328,_286).

Call: findlast([3],_355,_286).
```

Nu har vi formen som motsvarar

```
Exit: findlast([3],[],3).
```

Listorna ”fylls på” med H i omvänd ordning som huvudet plockades av, dvs. först 2 följt av 1. När den får svar på exekvering 3 att H är 3. Kan den gå bakåt i exekveringsstacken och fylla i där den inte visste. Så den går tillbaks till att 3an är T. Då ovanför är [H | T] = 2 | 3 och [H | Rest] = [2].

```
Exit: findlast([2,3],[2],3).
```

Sen går den tillbaka en gång till och då är T = [2,3] och H är det som stod framför vilket är 1

```
Exit: findlast([1,2,3],[1,2],3).
```

Vilket skriver ut `R = [1,2]` och `E = 3.`

Uppgift 4

Definiera predikatet partstring/2 som givet en lista som första argument genererar en lista av ett antal element som man finner konsekutivt i den första listan!

Lösning:

```
partstring(L,R):-
    append(_,B,L),
    append(R,_,B).
```

Följande output ges:

```
?- partstring([1,2,3], R).

R = [] ? ;

R = [1] ? ;

R = [1,2] ? ;

R = [1,2,3] ? ;

R = [] ? ;

R = [2] ? ;

R = [2,3] ? ;

R = [] ? ;

R = [3] ? ;

R = []
```

Beskrivning av exekvering:

Först anropas predikatet `partstring([1,2,3], R).`

Det prolog gör då är att reservera en minnesadress för R som följer

```
Call: partstring([1,2,3],_285)
```

Sedan anropas `append(_,B,L),`

```
Call: append(_316,_356,[1,2,3])
```

Detta returnerar `B = [1,2,3].`

Återigen anropas append, denna gång med `(R, _, B).`

Vilket enligt appends natur returnerar

```
Exit: append([],[1,2,3],[1,2,3])
```

Det vill säga `R = [ ].`

Nu är ”ett varvs rekursion” avklarat, vilket skriver ut `R = [ ].` i terminalen.

Genom backtracking testas en alternativ unifiering, denna gång med partstring([1,2,3], [ ]).

```
Redo: partstring([1,2,3],[ ]) ?
Redo: append([ ],[1,2,3],[1,2,3]) ?
Exit: append([1],[2,3],[1,2,3]) ?
Exit: partstring([1,2,3],[1]) ?
```

Detta upprepas tills dess att rekursionen terminerar då append anropas med bara tomma listor.

Uppgift 5

Definiera predikatet permute/2 som givet en lista som första argument genererar en lista av alla elementen man finner i den första listan ordnad på något av de möjliga sätten!

Lösning:

```
permute([ ],[ ]).
permute([H|T],Y):-
    permute(T,L),
    select(H,Y,L).
```

Följande output ges:

```
| ?- permute([1,2,3], Y).

Y = [1,2,3] ? ;

Y = [2,1,3] ? ;

Y = [2,3,1] ? ;

Y = [1,3,2] ? ;

Y = [3,1,2] ? ;

Y = [3,2,1] ? ;
```

Beskrivning av exekvering:

Först anropas predikatet `permute([1,2,3], Y).`

Det prolog gör då är att reservera en minnesadress för Y som följer

Call: permute([1,2,3], _394)

permute anropas rekursivt med svansen på listan och nya adresser för Y.

```
Call: permute([2,3],_463)
Call: permute([3],_487)
Call: permute([],_511)
Exit: permute([],[])
```

Predikatet `select` anropas med det senaste huvudet i listan (3), ny adress för Y och L som från tidigare rader unifierats till [ ].

```
Call: select(3, _537, [ ])
```

Nu går den tillbaka i exekveringsstacken och ser att Y = [3].

```
Exit: select(3,[3],[])
```

```
Exit: permute([3],[3])
```

Select körs igen, denna gång med det som stod framför 3, dvs. 2. Nu är det känt att L = [3].

```
Call: select(2,_566,[3]) ?
```

Återigen, går tillbaka och ser att Y = [2,3]. L är fortfarande = [3].

```
Exit: select(2,[2,3],[3])
```

```
Exit: permute([2,3],[2,3])
```

Upprepas sedan för 1 som följer

```
Call: select(1,_394,[2,3])
```

```
Exit: select(1,[1,2,3],[2,3])
```

Nu vi vi svaret på vår ursprungliga fråga, dvs Y = [1,2,3].

```
Exit: permute([1,2,3],[1,2,3])
```

Genom backtracking fås de fem andra svaren på ett liknande sätt, dock behöver Prolog inte göra några nya ”calls” i backtracking, eftersom den kan använda det den vet från ovanstående.