Logic Programming

Rodica Potolea
Camelia Lemnaru

Lecture #4

Cluj-Napoca

Agenda

- Forward vs backward recursion
  - sum
  - generics
  - Comparative analysis, Pros and cons
- append3
  - Forms
  - Efficiency analysis and justification
  - Nondeterministic call
- Delete from a list
  - One, just one, all, repeating the query

Forward vs backward recursion

- Forward vs backward - alternative approaches to process data
- Forward = processing takes place when the current item is first encountered, and the rest of the data (the other components than the item) are processed AFTER the current item is processed.
- Backward = starts by processing all but current item (the other components than the item) via recursive call(s) while the current item is processed just AFTER we return from recursion(s). NOTE: is returned from recursive call (and **NOT from backtracking!!!**).
- In case of lists [H|T]:
  - *forward* handles H first and next T (via recursive call),
  - *backward* starts by solving the problem on T (via recursive call), and just when returning from the call H is processed.

Forward sum

```
% sum_2/3

%sum_2(in_list, final_result, accumulator_part_result,).

sum_2([],PartialSum,PartialSum).    //final result, arg 2, copies the value of the partial result, arg3
                                    //with default unification
sum_2([H|T],Sum,PartialSum):-
        NewPartialSum is PartialSum + H,   //do process the current item
        sum_2(T,Sum,NewPartialSum).        //go ahead with the reminding structure
```

- List decomposed into [H|T]
  - Starts by processing (addition here) the current item (H here)
  - Continue with processing the rest of the structure (one recursive call here, as partition is H and T)
  - This implies the items are added in the sum forward (starting from the first one).
- ->->->…-> processing is performed in the order of items in the structure
- The accumulated result represents the sum of values from the beginning to the current item

Backward sum

```
% sum_1/2
%sum_1(in_list,sum_of_els_in_arg_1).
sum_1([],0).                    //result gets initialized. Empty input, null output
sum_1([H|T],Sum):-
        sum_1(T,TailSum),       //call first the processing on the rest of the partition
        Sum is TailSum + H.     //do process the current item
```

- List decomposed into [H|T]
  - Starts with processing T, same predicate, recursive call
  - When returned, use the obtained result (TailSum) to evaluate the overall result on the whole list (Sum)
  - This implies the items are added in the sum backwards (starting from the last one).
- ---… --> first go this way (all way to the end) doing nothing (just call)
- <- the processing occurs after returning from the call, one at a time
- <- in the opposite order, *last* item *first* processed, before last item second processed, ….
- <- third, second and first items being last processed in this specific order (first is last)
- The accumulated result represents the sum of values from the current item to the end

Forward vs backward sum

- Forward solution needs specific initial call => write a special predicate (a wrapper) to make it in a sound way (avoid the need of user knowing how to initialize the accumulator parameter) :

```
run_sum_2(List,Sum):-                //same partial result as in case of backward
        sum_2(List,Sum,0).           //nothing yet processed, null result. Same as
                                     //in stop condition for bwd.
```

- If the input list is [1,2,3,4,5,6,7] what is going to be the partial result (PartialSum) and (TailSum) respectively on forward and backward solutions?
- Try to **estimate before running** them.
- Follow the textbook to update the predicate with the necessary lines to print the values.

Forward generic

%forward_recursion/3

%forward_recursion(input argument, final result, partial result)

forward_recursion([],PartialResult,PartialResult). //final result, arg 2, copies the //value of the partial result, arg3, with default unification

forward_recursion([H|T],Result,PartialResult):- //partition data, here split into H and T

do(NewPartialResult,H,PartialResult), //start by processing the current item //and thus updating the previous PartialResult to NewPartialResult via processing do.

forward_recursion(T,Result,NewPartialResult) //process the rest of the structure //with recursive call

%forward_recursion_call/2

%forward_recursion_call(in, out)

forward_recursion_call(Input,Output):-

forward_recursion(Input,Output,InitialValueOfResult) //make the initialization with //a separate predicate (wrapper) to avoid mandatory user initialization

Backward generic

```
%backward_recursion/2
%backward_recursion(input argument, output result)


backward_recursion([],InitialValue).        //empty input, make initialization backwards
backward_recursion([H|T],PartialResult):-
        backward_recursion(T,NewPartialResult), // starts with processing the rest of
                                  //the structure; all partition but the current item.
        do(PartialResult,H,NewPartialResult).    //process the current item
```

- No need for a specific initial call, hence, no wrapper predicate.

Forward vs backward pros and cons

	Forward	Backward
+	• Process "as we go" => structure is processed from front to end => intermediate results could be useful (is the result of the structure "so far"). • In concurrent processing that result is made available and another process using it can start immediately • Last call optimization = reusing the same stack area without the need of restoring it back	• Needs no initialization=>Needs no specific call => needs no wrapper => needs no additional argument
-	• Needs specific initial call => always make a wrapper to initialize the accumulator • Needs additional argument (partial result)	• Intermediate results are seldom useful • The result of the structure is known just at the end of the processing, so, concurrency is postponed on sync

Concatenate 3 lists

- Use what you ***have*** vs use what you ***know***
- We have the concatenation of 2 lists
- Use it twice.

```
append([],List,List).

append([Head|Tail],List,[Head|Rest]):-

        append(Tail,List,Rest).
```

- To put together L1, L2 and L3 do the following
  - (L1+L2) + L3

  OR
  - L1 + (L2+L3)

Concatenate 3 lists: Use what you have 1

- (L1+L2) + L3; say |L1|=n1, |L2|=n2, |L3|=n3,

```
append([],List,List).                       //order of classes does not matter
append([Head|Tail],List,[Head|Rest]):-      //due to the indexation
        append(Tail,List,Rest).             //on the first argument
```

- Efficiency: O(n) where n the length of the first parameter

```
append3_1(L1,L2,L3,Result):-
        append(L1,L2,Intermediate),         //link L2 at the end of L1
        append(Intermediate,L3,Result).     //link L3 at the end of the
                                            //intermediate result created before.
```

- Efficiency:
  - O(n1) for the first call (links L2 at the end of L1)
  - O(n1+n2) for the second call, length Intermediate of is length of L1 and L2 (links L3 at the end of Intermediate)
  - Overall: t(n)=2n1+n2

Concatenate 3 lists: Use what you have 2

- L1 + (L2+ L3); say |L1|=n1, |L2|=n2, |L3|=n3,

```
append3_2(L1,L2,L3,Result):
        append(L2,L3,Intermediate),          //link L3 at the end of L2
        append(L1,Intermediate,Result).      //link result created before
                                 //Intermediate at the end of the first list L1
```

- Efficiency:
  - O(n2) for the first call, decomposes L2 (links L3 at the end of L2)
  - O(n1) for the second call, decomposes L1 (links Intermediate at the end of L1)
  - Overall: t(n)=n1+n2
- Observations:
  - **Second version better regardless the input!**
  - Order of calls matters (again!)
  - How can we use this in a standalone predicate?

Concatenate 3 lists: Use what you know

- What we know? Concatenation via decomposition of the first argument! Use it!

```
append3_3([Head|Tail],List2,List3,[Head|Rest]):- //as long as the first arg
        append3_3(Tail,List2,List3,Rest).    // nonempty, decompose it
append3_3([],[Head|Tail],List,[Head|Rest]):-//once first argument empty
        append3_3([],Tail,List,Rest).//you are back on 2 list concatenation
append3_3([],[],List,List).
```

- Observations:
  - Clauses 2 and 3 are disjoint from clause 1 (indexation on the first argument would treat them separately, i.e. cl1 one entrance, cl 2 and 3 another one). Therefore, it does NOT matter where clause 1 is placed
  - On the other hand, clause 3 should come AFTER clause 2 (as indexation on the second argument is NOT available

Concatenate 3 lists: Use what you know – contd.

```
append3_3([Head|Tail],List2,List3,[Head|Rest]):-
        append3_3(Tail,List2,,List3,Rest).   // decomposes first list
append3_3([],[Head|Tail],List,[Head|Rest]):-
        append3_3([],Tail,List,Rest).        // decomposes second list
append3_3([],[],List,List).
```

- Observations:
  - How is done? (resembles in behavior to version1)
    - Decomposes L1 to go through it by adding each of its items in result (behaves as version 1, as in "link L2 at the end of L1")
    - Decomposes L2 to go through it by adding each of its items in result (links L3 at the end of L1 concatenated to L2).
  - Efficiency (resembles in performance to version2) t(n)=n1+n2
    - O(n1) first clause
    - O(n2) second clause

Concatenate 3 lists: Use what you know – contd.

- Behavior: resembles to version1 (L1+L2) + L3
- Efficiency: resembles to version2 $t(n)=n1+n2$
- How is possible? Behavior V1 and performance V2?
- Explain!
- Explain which of the 3 version is best?
- Which to use and why?
- Which should never be used? Why?

Concatenate 3 lists: Nondeterministic call

- Given append3 predicate and the call: ?-append3(X,Y,Z,[1,2]).
- What is the meaning of the call?
  - Nondetermenistic call
  - What are the lists X,Y and Z whose concatenation form list [1,2].
- What is/are the result/s?

X	Y	Z
[]	[]	[1,2]
[]	[1]	[2]
[]	[1,2]	[]
[1]	[]	[2]
[1]	[2]	[]
[1,2]	[]	[]

Other results? Why not?

- Which order/why?
  - Depends on the implementation.
  - Identify (BEFORE running) the order of results in each of the 3 implementations
  - For append3 with append, the order of clauses on append is MANDATORY (fact first), otherwise it enters infinite loop with 3 free variables on the first call.

Delete from a list

- Given a list, remove one item from it.
- How many arguments/why?

```
%delete/3

%delete(item to remove, input list, output list)

delete(H,[H|T],T).//if the item to remove is head of input, just don't put it on output
delete(X,[H|T],[H|R]):- //otherwise, keep it on output and
        delete(X,T,R). //remove from tail
```

- What are the results on call when the item occurs several times? Try to estimate and justify BEFORE running.

  q1?- delete(3,[1,3,2,3,4,3],Output).

- What happens in case the item is not present in the list?

  q2?- delete(5,[1,3,2,3,4,3],Output).

Delete from a list – contd.

```
delete(H,[H|T],T).
delete(X,[H|T],[H|R]):-
            delete(X,T,R).
```

- What are the results on call? Why?

```
| ?- delete(1,[1,2,1,3,1],R).
R = [2,1,3,1] ? ;
R = [1,2,3,1] ? ;
R = [1,2,1,3] ? ;
no.
```

- What about the call? Why?

```
| ?- delete(4,[1,2,1,3,1],R).
no.
```

- So, the meaning of the predicate is: **delete exactly one occurrence** of an item from the list.

Delete from a list - contd

```
delete(H,[H|T],T).          //if the item to remove is head of input, just don't put it on output
delete(X,[H|T],[H|R]):-              //otherwise, keep it on output and
            delete(X,T,R).           //remove from tail
delete(_,[],[]).            //when the empty list is reached, whatever element is assumed to
                            //be deleted, done, result empty
```

- What are the results on call when the item occurs several times? Try to estimate and justify **BEFORE running**.
- What happens in case the item is not present in the list?

Delete from a list - contd

```
delete(X,[X|T],T).
delete(X,[H|T],[H|R]):-
            delete(X,T,R).
delete(_,[],[]).
```

- What are the results on call?

```
| ?- delete(1,[1,2,1,3,1],R).
R = [2,1,3,1] ? ;
R = [1,2,3,1] ? ;
R = [1,2,1,3] ? ;          //what's next? Why?
R = [1,2,1,3,1] ? ;
no.
```

- What about the call? Why?

```
| ?- delete(4,[1,2,1,3,1],R).
R = [1,2,1,3,1] ? ;
no.
```

- So, the meaning of the predicate is: **delete one occurrence** of an item from the list; if absent, do NOTHING (leave the list unchanged).

Delete from a list - contd

- What are the differences when the 2 implementations (without/with 3rd clause) are compared? Explain!
- What happens if the clause when item is found cuts the backtrack?
- Implementation without 3rd clause:

```
delete(H,[H|T],T):-!.
delete(X,[H|T],[H|R]):-
        delete(X,T,R).
```

- A cut in a clause is as if in all consequent clauses we add the negation of the conjunction to the left of the cut. Therefore, in a clause like:

  p:-**q,r**,!,s.

  The cut implies a "default" negation of **q,r** (therefore (**not(q,r)**)) in all clauses after it.
- In the predicate delete, first clause contains **nothing** to the left of the cut. So, what does it negate?
- Does it make any sense to place that cut?

Delete from a list - contd

```
delete(H,[H|T],T):-!.              //means          delete(X,[H|T],T):-X=H,!,
delete(X,[H|T],[H|R]):-            //therefore a     X<>H here
        delete(X,T,R).
```

- In the implementation with 3rd clause:

```
delete(H,[H|T],T):-!.
delete(X,[H|T],[H|R]):-
        delete(X,T,R).
delete(_,[],[]).
```

- Answer the same queries for **both** implementations and estimate the output (initial queries and repetitions). Explain!

```
| ?- delete(1,[1,2,1,3,1],R1).
R1=…? ; ? WHY?
| ?- delete(4,[1,2,1,3,1],R2).
R2=…? ; ? WHY?
```

Delete all occurrences of an item from a list

Start from the first version of the predicate

```
delete(H,[H|T],R):-     //if the item to remove is head of input, don't put it on output
        delete(H,T,R).       //but also remove other occurrences from tail
delete(X,[H|T],[H|R]):-     //otherwise, keep it on output and
        delete(X,T,R).       //remove from tail
```

- Could it be just this?
- Justify!

Delete all occurrences of an item from a list

```
delete_all(H,[H|T],R):-//if the item to remove is head of input, don't put it on output
        delete_all(H,T,R).     //but also remove other occurrences from tail
delete_all(X,[H|T],[H|R]):-    //otherwise, keep it on output and
        delete_all(X,T,R).     //remove from tail
delete_all(_,[],[]).           //when the empty list is reached, done, result empty
```

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

- What happens if we repeat the query? Why?
- How many answers? Which order? Explain!
- How can we obtain just the answer from above?

Delete all occurrences of an item from a list

```
delete_all(H,[H|T],R):-!,
            delete_all(H,T,R).
delete_all(X,[H|T],[H|R]):-
            delete_all(X,T,R).
delete_all(_,[],[]).
```

- A cut in a clause is as if in all consequent clauses we add the negation of the conjunction to the left of the cut. Therefore, in a clause like:

  p:-**q,r**,!,s.

  The cut implies a "default" negation of **q,r** (therefore (**not(q,r)**)) in all clauses after it.
- In the predicate delete_all, first clause contains **nothing** to the left of the cut. So, what does it negate?
- Does it make any sense to place that cut?

Delete all occurrences of an item from a list

```
delete_all(H,[H|T],R):-!,
        delete_all(H,T,R).
```

What does ! negates?

Is the default unification! The clause above is like:

```
delete_all(X,[H|T],R):-
        X=H,!,
        delete_all(X,T,R).
```

Therefore, the cut negates X=H, thus, in the next clauses, there is a default X<>H.

Delete all occurrences of an item from a list

```
delete_all(H,[H|T],R):-!,
        delete_all(H,T,R).
delete_all(X,[H|T],[H|R]):-
        delete_all(X,T,R).
delete_all(_,[],[]).

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

- What is the result? Why?
- What happens if we repeat the query? Why? How many answers? Which order? Explain!

Conclusion

- Order of clauses matters
- Order of calls matters
- Always estimate performance
- Meaning of
  - Repeating the query
  - The cut
- Questions?

References explained

1. **Lloyd**, J. W, Foundations of Logic Programming, Springer, 1984
2. **Clocksin**, William, **Mellish**, Christopher S., Programming in Prolog - Using the ISO Standard, Springer 1981
3. Leon S. **Sterling** and Ehud Y. **Shapiro**, The Art of Prolog Advanced Programming Techniques
4. Robert **Kowalski**, Logic for Problem Solving, 1979
5. Rodica **Potolea**, Programare Logica (vol 1), UTPres, 2007