Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III
Prof. Dr. A. B. Cremers

Advanced Logic Programming
Summer Semester 2012
Dr. Günter Kniesel

Assignment 6

Due: Sunday, 10.06.2012, 23:59:59 via SVN

Task 1. *Output and backtracking-driven iteration* (2 Points)

a) Assume there is a predicate p/2. How would you print out all results for the call p(X,Y)? Write a query that does this.

```
?- p(X,Y), ...  .  % ← fill in the ...
```

Sample Solution:

```
?- p(X,Y), write(p(X,Y)),nl,fail.
```

you will notice that the query fails always. To make it true after it has printed all possible results, you can introduce a new predicate; call it mylisting/0.

```
mylisting:- p(X,Y), write(p(X,Y)),nl,fail.
mylisting. % true always
```

now invoke

```
?- mylisting.
```

Task 2. *Shifting lists* (3 Points)

In task 5.3 you developed a first version of a linearised list that mostly looked like the following "linear_simple/2":

```
linear_simple([],[]) .
linear_simple([X|L1],[X|L2]) :- keep_unchanged(X),
                                linear_simple(L1,L2).
linear_simple([X|L1],L4) :- \+ keep_unchanged(X),
                                linear_simple(X,L2),
                                      linear_simple(L1,L3),
                                      append(L2,L3,L4).

keep_unchanged(X):- X \== [], (atomic(X);var(X)).
```

Your new task is to write a version of linearisation, "linear_acc(+**L**,?**Res**)" that does not need the expensive call to "append/3". It should use a ternary helper predicate "linear_acc(+**L**,+**Acc**,?**Res**)" whose additional parameter **Acc** acts as an accumulator of the already linear part of the list. New elements should always be inserted in front of the accumulated intermediate result (not appended at the end).

Tip: Start by determining the proper initialisation value for the accumulator argument and then work out how you can "grow" it as described above.

```
% sample solution
/**
* linear_acc(+NestedList, ?FlatList)
*
* Succeeds whenever Arg1 is a list whose elements may
* themselves be arbitrarily deeply nested lists and Arg2
* contains all elements inside Arg1 in the same order but
```

```
* without any nesting.
*
* Uses linear_acc/3 which itself uses an accumulator to save
* all former seen elements.
*/
linear_acc(L,Res) :- linear_acc(L,[],Res).
/**
* linear_acc(+NestedList, +Accumulator, ?FlatList)
*
* Succeeds whenever
* - Arg1 is a list whose elements may themselves be arbitrarily
* deeply nested lists,
* - Arg2 contains a list without any nesting and
* - Arg3 contains all elements inside Arg1 in the same order but
* without nesting followed by all elements of Arg2.
*/
linear_acc([],Acc,Acc).

linear_acc([X|L],Acc,[X|Lnew]):-
     keep_unchanged(X),
     linear_acc(L,Acc,Lnew).

linear_acc([X|L],Acc,Lnew):-
     \+(keep_unchanged(X)),
     linear_acc(L,Acc,Lhelp),
     linear_acc(X,Lhelp,Lnew).

keep_unchanged(X):- X \== [], (atomic(X);var(X)).
```

Task 3. *Accumulators: Performance evaluation* (2 Points)

Compare the runtime of the simple and accumulator-based version from Task 1 by running the following test code 10 times and averaging the results for each version:

```
?- make_list(150,5,L),
   time( linear_acc (L,_) ),
   time( linear_simple(L,_) ).
```

The output will have the following structure:

```
% ... inferences, ... CPU in ... seconds (100% CPU, ... Lips)
% ... inferences, ... CPU in ... seconds (100% CPU, ... Lips)
L = ... the first elements of the generated test list...
```

Here, „inferences“ is the number of resolution steps performed, „CPU“ is the CPU time in milliseconds and „Lips“ is the abbreviation for „Linear Inferences per Second“ (inferences/CPU*1000). Hand in the above lines for all 10 test runs along with the average inferences and CPU time for each of the two predicate versions.

Tip: The above test uses the following helper predicates:

```
     /**
      * make_list(+Elems,+NestedElems,?List) is det
      *
      * Arg3 is a list with Arg1 random elements that are
      * either integers or lists created with make_list/3.
```

```
 * Arg 2 is the length of nested lists (at every
 * level of nesting.
 */
make_list(0,_,[]) :- !.
make_list(I,IS,[E|T]) :-
      make_head(IS,E),
      I_1 is I-1,
      make_list(I_1,IS,T).

/**
 * make_head(+L,?Res) is det
 *
 * Res is either
 *   - a random integer value between 0 and 100000 or
 *   - a list of length L that can itself have nested
 *     sublists of length L.
 * The choice between the two options is made randomly
 * with a probability of 20% (1 of 5) for the list case.
 */
make_head(I,E) :-
    NestIt is random(5),
    make_list_or_int(NestIt,I,E).

make_list_or_int (1,I,E) :- !, make_list(I,I,E).
make_list_or_int (N,_,E) :- E is random(100000).
```

Sample solution:

The results differ widely because the lists on which the flattening is tested can be very different (because they were randomly generated). However, the accumulator version is much faster and needs fewer inferences. The simple version often runs out of global stack.

Task 4. *Cuts* (2 Points)

a) Consider the following program

```
p(1).
p(2):-!.
p(3).
```

Write all answers to the following queries:

```
1  ?- p(X).
2  ?- p(X), p(Y).
3  ?- p(X), !, p(Y).
```

Sample solution:

```
?- p(X).
   X = 1 ;
   X = 2.

?- p(X), p(Y).
```

```
X = 1,
Y = 1 ;
X = 1,
Y = 2 ;
X = 2,
Y = 1 ;
X = 2,
Y = 2.
```

```
?- p(X), !, p(Y).
X = 1,
Y = 1 ;
X = 1,
Y = 2.
```

b) Explain for each case why the program gives that answer.
Sample solution:

a. The first and second clauses are tried. At the end of the second clause is a cut. So no alternative clause is tried.
b. p(X) is the same as query 1. After each decision for a value for X in p(X) the interpreter searches for an answer for p(Y) (with X bound). For each possible X we try the same possibilities for Y and reach the cut after we tried Y=2. We again try no other clauses for p(Y).
c. After our first decision for a value for X we see a cut in the query. So we fix variables we have already unified (X=1). After that we only try different cases for everything after the cut (p(Y)). Here again we stop after Y=2 for the same reason as above.

Tip: As a preparation for the oral exam at the end of the course do it with *pen and paper only*, not by running the program! Or even better: Try to explain it to a colleague. For oral exams, practicing talking is the best preparation! Run it and use the debugger only if you cannot do it with pen and paper!!!

Task 5. *Declarative and Procedural Meaning and Cuts* (3 Points)

Give the declarative and procedural meaning for the predicates p, q, r. Describe the cause for differences and similarities. Recall that the declarative meaning of cut is simply “true” (it always succeeds).

```
p:- a, b.
p:- c.

q:- a, !, b.
q:- c.

r:- c.
r:- a, !, b.
```

Sample solution:

p: declarative: Succeeds, iff a and b are both true OR c is true (and a and b may or may not be true).
procedural: This is either a AND b together OR c.
q: declarative: Succeeds, iff a and b are both true and c is false OR a is false and c is true (b may or may not be true).
procedural: If a, then b, else c (see slides).
r: This is similar to p, but the order of the clauses has changed. The cut does not affect the solution set. Therefore the procedural meaning is c OR (a AND b together) and the declarative succeeds, iff c is true OR a AND b are both true.

abc	**p**	**q**	**r**
000			
001	x	x	x
010			
011	x	x	x
100			
101	x		x
110	x	x	x
111	x		x

Fig1: Truthtable for p,q,r,a,b,c.

This Task 5 was inspired from the chapter about red cuts from the book of Bratko, who writes:[1]

[1] p139f, Bratko, Ivan "Prolog – Programming for artificial intelligence" Second Edition Addison-Wesley, 1990

The reservations against the use of cut stem from the fact that we can lose the valuable correspondence between the declarative and procedural meaning of programs. If there is no cut in the program we can change the order of clauses and goals, and this will only affect the efficiency or termination of the program, not the declarative meaning. On the other hand, in programs with cuts, a change in the order of clauses may affect the declarative meaning. This means that we can get different results. The following example illustrates:

```
p :- a, b.
p :- c.
```

The declarative meaning of this program is: p is true if and only if a and b are both true or c is true. This can be written as a logic formula:

```
p <==> (a & b) v c
```

We can change the order of the two clauses and the declarative meaning remains the same. Let us now insert a cut:

```
p :- a, !, b.
p :- c.
```

The declarative meaning is now:

```
p <==> (a & b) v (~a & c)
```

If we swap the clauses,

```
p :- c.
p :- a, !, b.
```

then the meaning becomes:

```
p <==> c v (a &  b)
```

The important point is that when we use the cut facility we have to pay more attention to the procedural aspects. Unfortunately, this additional difficulty increases the probability of a programming error.

In our examples in the previous sections we have seen that sometimes the removal of a cut from the program can change the declarative meaning of the program. But there were also cases in which the cut had no effect on the declarative meaning. The use of cuts of the latter type is less delicate, and therefore cuts of this kind are sometimes called 'green cuts'. From the point of view of readability of programs, green cuts are 'innocent' and their use is quite acceptable. When reading a program, green cuts can simply be ignored.

On the contrary, cuts that do affect the declarative meaning are called 'red cuts'. Red cuts are the ones that make programs hard to understand, andthey should be used with special care.

Task 6. *Sorting and Cuts* (6 Points)

Where are good places to use cuts to optimise the following sorting programs? What would be changed by the cuts?
Which version (a, b, c and each either with or without cuts) would you recommend to use? Explain the reason behind your recommendation. Implement your preferred version and give it a comprehensive description.

a)
```
sort([],[]).
sort([X],[X]).
sort([H|T1],[H,S|T2]):- sort(T1,[S|T2]), H < S.
sort([H|T1],[S|T3])  :- sort(T1,[S|T2]), not(H<S), sort([H|T2],T3).
```

b)
```
sort1([X|Xs], Ys) :- sort1(Xs, Zs), insert(X, Zs, Ys).
sort1([],[]).

insert(X, [], [X]).
insert(X, [Y|Ys], [Y|Zs]  ):- X > Y, insert(X, Ys, Zs).
insert(X, [Y|Ys], [X,Y|Ys]):- X=<Y.
```

c)
```
sort2([X|Xs], Ys):-
    partition(Xs, X, Littles, Bigs),
    sort2(Littles, Ls),
    sort2(Bigs, Bs),
    append(Ls, [X|Bs], Ys).
sort2([],[]).

partition([X|Xs], Y, [X|Ls], Bs):- X=<Y, partition(Xs, Y, Ls, Bs).
partition([X|Xs], Y, Ls, [X|Bs]):-  X>Y, partition(Xs, Y, Ls, Bs).
partition([], Y, [], []).
```

Sample solution:

```
/**
* sort_cut(+UnsortedList, ?SortedList)
*
* succeeds iff Arg1 is a list and Arg2 contains the same
* elements as Arg1 in ascending order.
* It is a kind of bubble sort.
*/
sort_cut([],[]).
sort_cut([X],[X]).
sort_cut([H|T1],[H,S|T2]):-
      sort_cut(T1,[S|T2]), H < S, !.
sort_cut([H|T1],[S|T3]) :-
      sort_cut(T1,[S|T2]), not(H<S),sort_cut([H|T2],T3), !.


/**
* sort1_cut(+UnsortedList, ?SortedList)
*
* succeeds iff Arg1 is a list and Arg2 contains the same
* elements as Arg1 in ascending order. The predicate uses
* the predicate insert/3 to accomplishe the task.
* It is a kind of insert sort.
*/
sort1_cut([X|Xs], Ys) :-
    sort1_cut(Xs, Zs), !,
    insert(X, Zs, Ys).
```

```
sort1_cut([],[]).


/**
* insert(+Elem, +SortedList, -NewList)
*
* succeeds iff Arg3 is the same List as Arg2 but with
* Arg1 added at its orderd position.
* Arg2 has to be a sorted List.
*/
insert(X, [], [X]).
insert(X, [Y|Ys], [Y|Zs]):-
    X > Y, !, insert(X, Ys, Zs).
insert(X, [Y|Ys], [X,Y|Ys]).



/**
* sort2_cut(+UnsortedList, ?SortedList)
*
* succeeds iff Arg1 is a list and Arg2 contains the same
* elements as Arg1 in ascending order. The predicate uses
* the predicate partition/3 to accomplishe the task.
* It is a kind of quick sort.
*/
sort2_cut([X|Xs], Ys):-
    partition(Xs, X, Littles, Bigs),
    sort2_cut(Littles, Ls),
    sort2_cut(Bigs, Bs),
    append(Ls, [X|Bs], Ys),!.
sort2_cut([],[]).

/**
* partition(+UnorderedKist +Pivot, -LeftList, -RightList)
*
* succeeds iff Arg3 contains all Elements of Arg1 that are
* smaller or equal than Arg2, Arg4 contains all Elements of Arg1 that
* are bigger than Arg2.
*/
partition([X|Xs], Y, [X|Ls], Bs):-
    X=<Y, !, partition(Xs, Y, Ls, Bs).
partition([X|Xs], Y, Ls, [X|Bs]):-
    partition(Xs, Y, Ls, Bs).
partition([], _, [], []).
```

Sort1_cut is the sort we would recommend to use. The time advantage of *quicksort* over *insertsort* is wasted in the above implementation because of the use of append. In Prolog you cannot access directly elements that are not at the top of a list (unless you use “difference lists”, which are not that easy to understand).