CMP-6040A/CMP-7028A Coursework hints

Pierre Chardaire

In this document I give a few hints to put you in the right direction. However, as one says, practice makes perfect and working on programming with Prolog, in particular going through the handouts, and the lab exercises and their solutions, should help you become better programmers. Make sure you understand how the bits of code I give here work.

Question1

The first question can be decomposed. It is clear a player can make two types of move, either by removing any number of stones from any one pile or any equal numbers of stones from any two piles. So first concentrate on writing a procedure (the several rules regarding a predicate that are grouped together) to remove any number of stones from any one pile. This can be decomposed into:

- Rules for removing from the first pile
  - Rule for Removing all stones from the first pile
  - Rule for Removing some but not all stones from the first pile
- Rule for removing from another pile

We have seen examples of predicates that use recursion on lists. For example in the 8-queen problem we had a procedure to delete from a list:

```
/*

 del(Item?, +L1, -L2)
 delete Item from L1 to produce L2

*/

del(Item,[Item|List],List).
del(Item,[First|List],[First|List1]):- del(Item,List,List1).
```

In this procedure the first rule, called base rule, deletes the first element in the list; the second rule deletes another element in the list. Note that the `between` predicate may be useful when you want to deal with "some but not all".

If you manage to reach this point with a procedure that works you are well on your way to complete the question.

For any two piles there is one that appears first in the state list and one that appears second (I am an expert on truisms!). As a consequence, removing any same number of stones from any two piles amounts to removing any number of stones from a pile in your state list and then removing *that same number* from any other pile that *appears further down the list*. This suggests first writing a procedure that removes *a given number* of stones from some pile. This procedure is quite similar to the one for the first type of move except that we have one extra input parameter which is the number of stones to remove. Obviously, you can remove a given quantity from a pile only if the pile has enough stones.

Once you have coded the procedure for removing a given number of stones from any one pile things should become easy. To remove from two piles then proceed as for removing from one pile (our first type of move), but modify your base case predicates to call the removing of a given number of stones from one of the subsequent piles in your state list to produce an updated list of subsequent piles.

Finally your move is simply a move of the first type or a move of the second type.

PGT question 2

I suggest the `count_play` predicate should involve a single rule that retracts all `counted` asserted facts and calls a `count_path` predicate that actually does the job. The number of paths from the final state `[]` is trivially one. The number of paths from a non-final state, S, is obtained by adding up the number of paths from each of the states that can be reached in one move from S. To do that I suggest using one of the standard predicates that collects all solutions in a list, in this cases these are states, and to process the list. Processing the list will involve checking if paths from a state have already been counted using the `counted` predicate and calling the `count_path` predicate if this is not the case. In the case of a call to `count_path` do not forget to memorise the result using an assert.

An accumulator-pair technique could be useful to do counting through a list. We have seen an example in the Prolog technique part of the module. Here is another example with lists

```
/*


sum(L+,N-)

Assumption L is a list of integers, possibly empty.

Computes N, the sum of the elements in L. N is 0 if L is empty.

*/

sum(L,N):-
    sum_acc(0,L,N).      % the Accumulator initial value is 0

sum_acc(Acc,[],Acc).     % Accumulator variable unified with returned variable

sum_acc(Acc,[H|T],Ret):-
    NewAcc is Acc + H,
    sum_acc(NewAcc,T,Ret).
```

UG question 2, PGT question 3

You should notice that the predicate `Win` has only one parameter which is a game state (position in the game). It does not specify the player who plays. Why? Because the game is a so-called symmetric game: it really does not matter who plays from that position as any one of the two players would play in the same way as the other player if players play the best move they can play.

In this game any position must be a winning position or a losing position as there is no tie. As a consequence a player who is *not* in a winning position is in a losing position. Now, a player is in a position of winning the game if she can make a move to a position where the other player will be in a losing position. Otherwise she is in a losing position. I cannot tell much more as this can be coded in exactly *four* lines of code.

This will work for small lists of small size piles, but it won't be very efficient as you will re-examine some states many times. However, you can vastly improve efficiency by using dynamic predicates to remember the status of positions already seen and, as a consequence, avoid re-exploration. In addition, there is scope for exploiting the fact that positions that are identical except for the order of their piles have the same status in terms of win/loss.

To code the more advanced version you may want to use the library predicate `msort` (use the help to see what it does). Also, we have seen some techniques of interest involving cuts in the labs. e.g.

```
is_prime(1):-!, fail.
is_prime(X):-
    Y is floor(sqrt(X)),
    between(2,Y,Z),
    0 =:= mod(X,Z),!, fail.

is_prime(_).


is_composite(X):-
   Y is floor(sqrt(X)),
   between(2,Y,Z),
   0 =:= mod(X,Z),!.

is_composite(_):-fail.
```

If you find the above code a bit difficult to understand why not trace it with small input values.

The rest

With the help given to start your coursework and c*areful reading and reflection* about my hints you should reach this point. The Undergraduate students should be able to do the rest without much help. Postgraduate students will have to think a little about what I am asking in question 6.b. . . . Anyway, I can't just tell you everything.