Solutions to Exercises for the Course

Logic

Week 8



Exercise 8.1 (Prolog and SLD Resolution)

a) For which queries `pp(...)` does the following Prolog program succeed? Provide an SLD resolution for `pp(f(f(f(a))))`.

```
pp(a).
pp(f(X)) :- pp(X).
```

b) What does the following predicate `what(A,B)` calculate when A is a list of numbers and B is a variable, e.g., `what([3,7,4,2,6],B)`?

```
what([H|T],I) :- what(T,I), I>H, !.
what([H|_],H).
```

Solution

a)

The query succeeds for all queries of the shape $\texttt{pp(f}^i\texttt{(a))}$ or $\texttt{pp(f}^i\texttt{(X))}$ for arbitrary nestings of $i \geq 0$ applications of `f`.

Here is an SLD resolution derivation for $pp(f(f(f(a))))$:

1. $pp(a)$ — program clause
2. $\neg pp(X)$, $pp(f(X))$ — program clause
3. $\neg pp(f(f(f(a))))$ — Query
4. $\neg pp(f(f(a)))$ — resolvent of 3 and 2 with $X = f(f(a))$
5. $\neg pp(f(a))$ — resolvent of 4 and 2 with $X = f(a)$
6. $\neg pp(a)$ — resolvent of 5 and 2 with $X = a$
7. $\square$ — resolvent of 6 and 1.

We see that each resolution step is between the query (in the first step) or the last generated clause (in the other steps) and one of the program clauses.

b)

For an empty list, `what([],I)` will fail, since this doesn't match the head of either of the two program clauses.

For a non-empty list, the predicate is first recursively evaluated on the tail of the list, giving `I`. The result is that number `I`, if it is greater than the head of the list.

Due to the cut, the second clause is only considered if the first one fails. This could be a) because the tail `T` is empty, or b) because `H` is greater or equal to `I`. In both cases, the second clause causes the head of the list is returned in `I`.

The result of this is that `what(A,B)` will compute the largest element of the list `A` and return it in `B`, and it will fail if `A` is empty.

But beware: `what([3,7,4,2,6],3)` will answer `yes`, see if you can figure out why.

Exercise 8.2 **(The Beauty of Prolog)**

a) Write a Prolog program `sum(L,S)` that calculates the sum `S` of a list `L` of numbers.

b) Write a one-clause Prolog program that calculates the minimum element of a list of numbers.

Solution

a)

The program has to recurse through the list. The main idea is that we have to be able to find

- the sum of the elements of an empty list, which is 0.
- the sum of the elements of a list of length $n + 1$ based on the length of a list of length n. Which we do by taking out one element, summing the others, and adding the element to the result.

```
sum([],0) .
sum([H|T],Sum) :- sum(T,Sum1) , Sum is Sum1 + H .
```

b)

Clearly, the program has to recurse through the list in order to find the minimum element. So there have to be two clauses, one for the recursion, and one for the base case.

How can we do this with only one clause? We can't really, so this has to be a trick question – most appropriately answered using a trick.

We give two solutions, using two different tricks.

Here is a solution that uses the "if-then-else" operator to separate the base case from the recursion case:

```
minel([H|L],Res) :- L=[] -> Res=H ; (minel(L,R), (H < R -> Res=H ; Res=R)) .
```

This is not typical Prolog code, it is much closer to the functional programming style. It is a one-liner, but of course it corresponds to more than one clause in Conjunctive Normal Form.

The other solution uses a built-in predicate that will do the recursion for us:

```
min(L,Min) :- member(Min,L), \+ (member(X,L), X<Min).
```

Remember that `\+` is the negation-as-failure operator, so the first `member` finds an element, and the negated part ensures that there is no smaller element. Again, this is not something that can be done with one clause of CNF. It relies on the negation as failure operator, and of course the built-in member predicate (which needs two clauses to define).

This answer is very much in the spirit of Prolog, in the sense that it says little more than what it means for an element to be minimal. It will have running time quadratic in the length of the list (because of what is essentially two nested loops for members of the list) but that is the price you pay for declarative programming without thinking about efficiency.

Remark: You can download SWI-Prolog from `http://www.swi-prolog.org/`.

Or GNU Prolog from `http://www.gprolog.org/`.