Constructive Logic (15-317), Fall 2017
Recitation 9: Logic programming

October 26, 2017

1 Logic programming

You might be familiar with functional and imperative programming. Today we will see yet another programming paradigm: logic programming. Logic programming can be seen as a fragment of intuitionistic logic[1] called *Horn clauses* (remember last homework?). A Horn clause is either an atom or a formula of the shape $A_1 \wedge \ldots \wedge A_n \supset H$, where H is called the *head* and $A_1 \wedge \ldots \wedge A_n$ is the *body*. In prolog syntax, this is written as:

```
h :- a1, a2, ..., an.
```

Let's step through a simple prolog program to understand how computation (or proof search) works. Consider the following simple code:

```
ocean_level(rising).
temperature(extreme).
global_warming(conspiracy) :- ocean_level(stable), temperature(normal).
global_warming(real) :- ocean_level(rising), temperature(extreme).
```

If we query prolog for `global_warming(X)`, it will look at the head of all (four) clauses trying to find one that "matches" (*unifies*) with the goal. In this case, it finds the clauses in lines 3 and 4. Prolog will process the options in order, so it will first go to clause in line 3 and unify `X` with `conspiracy`, generating the new goals `ocean_level(stable)` and `temperature(normal)`. A proof-theoretic interpretation of this step is the following (predicate names are abbreviated for the sake of space):

$$\dfrac{\dfrac{\text{ol(ris), temp(xtr), ...} \longrightarrow \text{ol(sta)} \quad \text{ol(ris), temp(xtr), ...} \longrightarrow \text{temp(nml)}}{\text{ol(ris), temp(xtr), ...} \longrightarrow \text{ol(sta)} \wedge \text{temp(nml)}} \wedge R \qquad \dfrac{\texttt{X is csp}}{\text{ol(ris), temp(xtr), gw(csp), ...} \longrightarrow \text{gw(X)}} \mathit{init}}{\text{ol(ris), temp(xtr), ol(sta)} \wedge \text{temp(nml)} \supset \text{gw(csp), ol(ris)} \wedge \text{temp(xtr)} \supset \text{gw(real)} \longrightarrow \text{gw(X)}} \supset L$$

In this derivation, X is a special variable that is unified on initial rules, and this unification propagates to the next branch if there were occurrences of X there as well. When trying to

[1] It is also a fragment of classical logic. Since it is such a simple fragment, intuitionistic and classical logic coincide.

prove the two open sequents, or the new goals, prolog will realize that `ocean_level(stable)` or `temperature(normal)` are not true... oops, are not in the context nor they are unifiable with any clause head. Time to backtrack. We know that $\wedge R$ is an invertible rule, so no use in backtracking there. We go back to the choice of clauses (i.e., $\supset L$) and try to use the one on line 4. This time the unification will be `X is real` and the new goals `ocean_level(rising)` and `temperature(extreme)`, which can be proved.

As a final note, logic programs hold some resemblance to functional programs in the way programs are written. You will find that sometimes the clauses used look a lot like the cases you would need in, say, SML. This kind of programming style is referred to as *declarative* programming (you write *what* your program does as opposed to *how* it does it).

Task 1. Implement a prolog program that computes the truncated subtraction between natural number along the same lines as the `plus` and `times` implementations given in the lecture notes.

```
pred(z,z).

pred(s(M), M).

minus(N, z, N).

minus(N, s(M), Q) :- minus(N, M, P), pred(P, Q).
```

In Prolog, lists are built in similarly to SML. The syntax for pattern matching on a list is `[Head | Tail]`. Using this we can implement a variety of programs for manipulating lists.

Task 2. Implement a prolog program which merges two sorted lists.

```
mymerge(L, [], L).
mymerge([], L, L).
mymerge([H1 | T1], [H2 | T2], [H1 | Out]) :-
    H1 =< H2,
    mymerge(T1, [H2 | T2], Out).
mymerge([H1 | T1], [H2 | T2], [H2 | Out]) :-
    H1 > H2,
    mymerge([H1 | T1], T2, Out).
```

Task 3. Implement a merge sorting procedure for lists.

```
split([], [], []).
split([X], [X], []).
split([H1 | [H2 | T]], [H1 | L1], [H2 | L2]) :-
    split(T, L1, L2).

mysort([], []).
mysort([X], [X]).
mysort([X1 | [X2 | L]], O) :-
    split([X1 | [X2 | L]], Left, Right),
    mysort(Left, SLeft),
    mysort(Right, SRight),
    mymerge(SLeft, SRight, O).
```

2 Modes

It's common in Prolog code to denote certain arguments to a relation as "inputs" and some as "outputs". These is the *mode* of an argument. An important property to ensure that your prolog programs terminate is to ensure that they are well-moded. That is, the inputs to a subgoal as well as the outputs are either determined by inputs or outputs of a previous goal. For instance, attempt to verify whether or not the following prolog programs are well-moded.

```
notprime(P) :-
  divisible(P, Q) %% divisible takes two inputs and holds when P % Q = 0

times(z, N, z).
times(s(M), N, O) :-
  times(M, N, U),
  plus(U, N, O).

% recall synth has the mode input, output
synth(inl(M), or(A, B)) :-
  synth(M, A).
```