Constructive Logic (15-317), Spring 2022
Recitation 8: Prolog Basics (2022-03-16)
clogic staff

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 called *Horn clauses*. A Horn clause is either an atom or a formula of the shape $A_1 \wedge ... \wedge A_n \supset H$, where H is called the *head* and $A_1 \wedge ... \wedge A_n$ is the *body*. In prolog syntax, this is written as:

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

Judgement in Prolog is represented as predicates. For example, if I say *Alice is a dog*, then the corresponding prolog representation will be

```
dog(alice).
```

where `dog` is a predicate, and `alice` is an atom[1]. Another example is *Pittsburgh is a city of Pennsylvania*, which can be represented as

```
city(pittsburgh, pennsylvania).
```

Judgement with a condition is represented using `:-`. For example, *Alice is Bob's daughter if Alice is Clare's daughter and Clare and Bob are couple* can be represented as[2]

```
daughter(alice, bob) :-
  daughter(alice, clare),
  couple(clare, bob).
```

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).
```

[1]The convention in Prolog is that names that start with a lower case letter (e.g `alice`) are constants (or atoms), and names that start with an upper case letter are unification variables

[2]The convention to refer to the predicate is the name followed by its arity (the number of arguments it takes). For example, we would say `dog/1`, `city/2`, `daughter/2`.

The first thing we could ask Prolog is *Is global warming a conspiracy?*

```
?- global_warming(conspiracy).
false.
```

Prolog said `false`. Then *Is global warming real?*

```
?- global_warming(real).
true.
```

If we query prolog for `global_warming(X)`[3], 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):

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

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 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).

2 Natural Number

Recall from Heyting arithmetic, we define natural numbers according to the following rules.

$$\dfrac{}{z : nat}\ natI_z \qquad\qquad \dfrac{n : nat}{sn : nat}\ natI_s$$

This can be encoded in Prolog as

```
nat(z).
nat(s(N)) :-
  nat(N).
```

[3]Here `X` is a unification variable, meaning that we are trying to find out the possible values for such variable.

Task 1. Define `plus/3` and `mult/3` where `plus(M, N, P)` satisfies that `M + N = P` and `mult(M, N, P)` satisfies that `M * N = P`.

Solution 1:

```
plus(z, N, N).
plus(s(M), N, s(P)) :-
  plus(M, N, P).

mult(z, N, z).
mult(s(M), N, P) :-
  plus(Q, N, P),
  mult(M, N, Q).
```

3 Modes

We often talk about modes in Prolog. Modes are a way of describing how a predicate will be used. We denote an argument of a predicate with + if that argument is provided to the predicate, and with - if that argument is outputted by the predicate. A defined Prolog predicate can often work with many different modes.

Note that not all possible modes work correctly for a given predicate in Prolog. Many predicates do not work with all arguments moded negatively. For example, given a predicate `permutation(L, P)`, where `L` is a list and `P` is a permutation of `L`, the mode `permutation(-L, -P)` will not terminate because there are infinite pairs `(L, P)` such that `P` is a permutation of `L`.

Take `plus/3` as an example. What are correct modes for this predicate? For a starter we could make the following queries.

```
?- plus(s(z), z, s(z)).
true.

?- plus(z, z, s(z)).
false.
```

This would correspond to `plus(+M, +N, +P)`, which returns a binary result true or false. We could also do the following:

```
?- plus(s(z), s(s(z)), X).
X = s(s(s(z))).
```

which corresponds to mode `plus(+M, +N, -P)`. We can push this idea even further:

```
?- plus(s(z), X, s(s(z))).
X = s(z).
```

This is effectively `minus`, which corresponds to mode `plus(+M, -N, +P)`.

Task 2. Prove that `plus(+M, -N, +P)` is a correct mode.

Solution 2: To prove a predicate has a certain mode, we show that for every clause of this predicate, if we know the input, then we can deduce the output. In this particular case, we proceed by induction.

For clause `plus(z, N, N).`, suppose we know `z` (which is uninteresting) and `N` (the second `N`), then we know `N` (the first `N`) trivially by the second input.

For clause `plus(s(M), N, s(P)) :- plus(M, N, P).`, suppose we know `s(M)` (then we immediately know `M`) and `s(P)` (then we immediately know `P`), we want to show that we also know `N`. By induction hypothesis: if we know `M` and `P` (which we do!), then we know `N`.

Task 3. Which modes does `mult` work correctly with?

Solution 3: Note that (+, +, -) does not work, even though it might be expected to, due to the infinite possibilities in the `plus(-, +, -)` mode.

- (+, +, +)
- (+, -, +)
- (-, +, +)

Task 4. How might we rewrite `mult` if we wanted it to work correctly with the `mult(+, +, -)` modality?

Solution 4: Change the order of the `plus` and `mult` in the third clause of the predicate.

4 Work With Lists

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.

Consider predicate `mem/2`, where `mem(X, L)` represents that *X is an element in L*. We could implemented it as followed.

```
mem(X, [Y | Ys]) :-
  X = Y.
mem(X, [Y | Ys]) :-
  X \= Y,
  mem(X, Ys).
```

The first clause can also be implemented as

```
mem(X, [X | _]).
```

Now we can run the queries.

```
?- mem(a, [b, c, a, d]).
true .

?- mem(a, [b, c, d]).
false.
```

Task 5. Define `listsum(L, N)` where `L` is a `nat` list, and the sum of all natural numbers in `L` is `N`.

Solution 5:

```
listsum([], z).
listsum([X | Xs], N) :-
  nat(X),
  listsum(Xs, M),
  plus(X, M, N).
```

5 A Complicated Example

Consider Untyped Lambda Calculus defined as followed[4].

$$M ::= x \mid \lambda x.M \mid ap(M, N)$$

We can encode those three constructs in Prolog respectively as `v(x)`, `lam(x, m)`, `app(m, n)`. The semantics of this language, perhaps unsurprisingly, is defined as

$$\frac{e_1 \to e_1'}{ap(e_1, e_2) \to ap(e_1', e_2)} \qquad \frac{}{ap(\lambda x.e, e_2) \to [e_2/x]e}$$

We are interested in a predicate `eval/2`, where `eval(E, F)` represents that term `E` evaluates to term `F`. We begin by exactly following the rules.

```
eval(app(E1,E2), F) :-
  eval(E1,E1prime),
  eval(app(E1prime,E2), F).

eval(app(lam(X,E),E2),F) :-
  subst(E2,X,E,F).
```

All it remains is to implement `subst/4`, where `subst(E1, X, E, F)` represents substituting `E1` into the variable of `X` in `E` results in `F`. How substitution works can be summarized as followed.

- $[r/x]x = r$
- $[r/x]y = y$ where $x \neq y$
- $[r/x]ap(e_1, e_2) = ap([r/x]e_1, [r/x]e_2)$

[4]Untyped Lambda Calculus is so elegant that it only has three construct: variable, function abstraction, and function application, yet it is equivalent to the Turing Machine, which defines the computability.

- $[r/x]\lambda x.e = \lambda x.e$
- $[r/x]\lambda y.e = \lambda y.[r/x]e$ where $x \neq y$

Now we an implemented it:

```
subst(E,X,v(X),E).

subst(E,X,v(Y),v(Y)) :-
  X \= Y.

subst(E,X,app(E1,E2),app(E1prime,E2prime)) :-
  subst(E,X,E1,E1prime),
  subst(E,X,E2,E2prime).

subst(E,X,lam(X,F),lam(X,F)).
subst(E,X,lam(Y,F),lam(Y,Fprime)) :-
  X \= Y,
  subst(E,X,F,Fprime).
```

Now this works almost as intended. However it is still wrong. Consider `subst(v(x), y, lam(x,v(y)), A).` Yikes. Then we need to add one more restriction to the last rule of substitution, which is that variable y is not a free variable in r (the variable y is said to be "fresh" for r). This would be hard to implement in Prolog though, so we left it as a bonus.