Lecture 11: Prolog

Timothy Gu

May 9, 2019

1 Prolog

Recall imperative and functional languages. Here's where Prolog and logic programming languages fit:

Paradigm	Example	Unit	Example
imperative	Java	commands	$S_1, S_2, S_3, \ldots$
functional	OCaml	functions	$f(g(x), h(y))$
logic	Prolog	predicates and relations	$(P(x) \land Q(x)) \lor P(y)$

Prolog has no traditional functions – it has predicates, which can be thought of as just functions that return a boolean. This is the hallmark of a *logic programming language*.

In Prolog, you declare what a correct solution looks like and Prolog searches for it. This makes Prolog a *declarative programming language*.

Prolog attempts to separate an Algorithm into Logic and Control, where Logic dictates correctness and Control dictates efficiency. The programmer writes logic, and the Prolog compiler figures out the mechanics on how to achieve that. Contrast this with Java, where the two are combined. Making something faster may introduce bugs.

2 A first example: sorting

`sort` We first define what it means for something to be sorted. Create a clause:

```
sort(L, S) :- sorted(S) , perm(L, S) .
```

S is a sorted version of *L* if *S* is sorted and *L* and *S* are permutations .

In a way, Prolog "if" is reversed compared to other programming languages; the predicate for "if" comes after the implication. But the similarity pretty much stops there.

`sorted` Now define `sorted`. A first attempt:

```
sorted([]). /* empty list is sorted */
sorted([_]). /* singleton list is sorted */
sorted([X, Y | L]) :- X =< Y, sorted(L).
```

Notes:

- `x, y | L` means `x :: y :: L` in OCaml.
- $\leq$ is spelled `=<`, and $\geq$ is spelled `>=`.

This is wrong! We only checked if the first two elements. Replace last line with:

```
sorted([X, Y | L]) :- X =< Y, sorted([Y | L]).
```

`perm` Now define `perm`.

```
perm([], []).
perm([X], [X]).
perm([X, Y], [X, Y]).
perm([X, Y], [Y, X]).…
```

Wait a sec, we need the recursive case... Replace last three lines with:

```
perm([X | L], R) :-
append(P1, [X | P2], R),
append(P1, P2, P),
perm(L, P).
```

Here, we essentially divide R into three pieces: P_1, $[X]$, and P_2. If this is possible then we recursive.

`append` Now define `append`.

```
append([], L, L).
append([X | L], M, [X | LM]) :- append(L, M, LM).
    /* If L ∪ M = LM, then ({X} ∪ L) ∪ M = {X} ∪ LM */
```

Of course, many of these predicates (including `sort` and `append`) are provided by Prolog itself. We are just doing this for instructive purposes.

Executing code. To run the code, Prolog has a prompt:

```
| ?- sort([3, -5, 7], R).
R = [-5, 3, 7].
```

Reordering predicates However, Prolog has a determistic behavior, going from left to right. Consider `sort(L, S)`; Prolog first considers `sorted(S)`, trying all possible S (starting with the empty list, a singleton list, etc.) so that it fits `perm(L, S)`.

A way to fix that is to just reverse the order of the predicates:

```
sort(L, S) :- perm(L, S), sorted(S).
```

This only changed the control – not logic, as the "and" operation is commutative.

Now the solution works.

Note, this algorithm first looks at all possible permutations of L – making this algorithm $O(n!)$ time. But generally Prolog is used to see if an algorithm is logically correct; if proven to be so then people would rewrite the code in some other language.

Also note, Prolog does not allow writing "not" easily. In this way, Prolog functions like BNF grammar productions, which has only "and"s and "or"s.

3 A second example: list member

Consider

```
member(X, [X | _]).
member(X, [_ | L]) :- member(X, L).
```

and

```
| ?- member(Q, [5, 3, 1]).
Q = 5 ?
```

Here, we have the option of letting Prolog give us a new valid answer (`;`), or stop (RET). If we input `;`, then we can get

```
| ?- member(Q, [5, 3, 1]).
Q = 5 ? ;
Q = 3 ? ;
Q = 1
yes
```

This feature could be described as automatic backtracking.

What if try to find a list that contains 9?

```
| ?- member(9, L).
L = [9|_] ? ; /* derived directly from the first rule */
L = [_,9|_] ? ; /* derived first from the second rule */
L = [_,_,9|_] ? ;
L = [_,_,_,9|_] ? …
```

This program enters an infinite loop – avoid writing programs like this!

What if we try to find a three-element list that contains 9?

```
| ?- member(9, [A, B, C]).
A = 9 ? ;
B = 9 ? ;
C = 9 ? ;
no
```

If we reject all of the possible predicates, the interpreter returns `no`.

With this insight, now consider how `append` works.

```
| ?- append([3, 1], [9], R).
    X1 = 3, L1 = [1], M1 = [9], R1 = [3 | LM1]
        append([1], [9], LM1)
            X2 = 1, L2 = [], M2 = [9], R2 = [1 | LM2] = LM1
                append([], [9], LM2)
                    L3 = [9] = LM2
R = [3, 1, 9]
```

We could also run `append` backwards:

```
| ?- append(L, M, [3, 1, 9]).
L = [], M = [3, 1, 9] /* due to first rule */ ? ;
    [3 | L1] = L, M1 = M, [X1 | LM1] = [3, 1, 9] ⟹ X1 = 3, LM1 = [1, 9]
        append(L1, M1, [1, 9])
            L1 = [], M1 = [1, 9] /* due to first rule */
L = [3], M = [1, 9] ? …
```

4 A third example: list reversal

```
reverse([], []).
reverse([X | L], R) :-
  reverse(L, J), /* J is L reversed */
  append(J, [X], R).
```

Recall from when we did such a function in OCaml, this runs in $O(n^2)$ time. Let's try rewriting this using the accumulator pattern:

```
revapp([], A, A).
revapp([X | L], A, R) :- /* {X} ∪ L ∪ A becomes R = J ∪ {X} ∪ A */
  revapp(L, [X | A], R). /* R = J ∪ {X} ∪ A */

reverse(L, R) :- revapp(L, [], R).
```

5 Prolog arithmetic

Consider

```
| ?- X = 2+3.
X = 2+3
```

The + actually constructs a data structure – a tree, with a root node of +.

To actually do arithmetic in Prolog, use the `is` primitive:

```
| ?- X is 2+3.
X = 5
```

which is syntactic sugar for

```
| ?- is(X, 2+3).
```

Note, we cannot include any logical variables in the right-hand side.