Prolog’s execution strategy explained more precisely

The execution strategy has been presented as simply placing all matching clauses onto the stack. In fact, Prolog is slightly more selective and a knowledge of this allows more efficient programs to be written.

Revision of the elementary Prolog data types

For this lecture we need to remember some key terms: functor, arity, and some Prolog data types:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom</td>
<td>cost, []</td>
</tr>
<tr>
<td>Number</td>
<td>1</td>
</tr>
<tr>
<td>Atomic</td>
<td>[], cost, 1</td>
</tr>
<tr>
<td>Compound object</td>
<td>car(ford, 1995)</td>
</tr>
<tr>
<td>List</td>
<td>[a, b, c]</td>
</tr>
<tr>
<td>Variable</td>
<td>Var</td>
</tr>
</tbody>
</table>

Prolog’s execution strategy revisited - 1

Consider this program:

- % 1 - terminating
  app([], List, List).
- % 2 - recursive
  app([Hd|L1], L2, [Hd|L3]) :-
    app(L1, L2, L3).

and the query:

| ?- app([a], [1], List). |

What goes on the stack?

Prolog’s execution strategy revisited - 2

Step 1 - add to stack

```
app([], List, List).
app([Hd|L1], L2, [Hd|L3]) :-
    app(L1, L2, L3).
```

Step 2 - first member of the stack is popped - but doesn’t match, so the next item on the stack is used

```
app([Hd|L1], L2, [Hd|L3]) :-
    app(L1, L2, L3).
```

Prolog’s execution strategy revisited - 3

Step 3 - The head of this rule matches, so the recursive subgoal is called - as if with the query:

| ?- app([], [1], List). |

The causes two more instances of the rule to be placed on the stack:

```
app([], List, List).
app([Hd|L1], L2, [Hd|L3]) :-
    app(L1, L2, L3).
```
Prolog’s execution strategy revisited - 4

Step 4 - The first member of the stack is popped - and this time matches - giving success and leaving the second clause on the stack:

\[
\text{app}([\text{Hd}|\text{L1}], \text{L2}, [\text{Hd}|\text{L3}]) :- \\
\text{app}([\text{Hd}|\text{L1}], \text{L2}, \text{L3}).
\]

But this is wasteful - we can tell in each step that one clause will never match either query!

Prolog’s indexing - or being almost as clever as us

Good Prolog implementations cut down the number of items added to the stack. The technique is simple:

All clauses are indexed by: functor arity type of the first argument

and, if the first argument is an atomic, then Prolog also indexes according to the value of the atomic.

For each query and subgoal, Prolog uses the index to identify eligible clauses.

What effect does Prolog’s indexing have on program design? - 1

Consider the following two queries:

| ?- app([a], [1], List). |
| ?- app([L1, L2, [a,1]]).

What goes on the stack for each of these queries?

What effect does Prolog’s indexing have on program design? - 2

From our knowledge of Prolog’s indexing, we can draw up the following guideline:

When designing a procedure:
- try to place the “input” arguments first;
- try to put the most discriminating in the first argument position.

Determinism

So far, we’ve defined determinism as having only one way of satisfying a goal, eg:

Answer is 2 + 3

can only be satisfied once.

Experienced Prolog programmers use a more subtle definition based on how many alternative clauses can be added to the stack.

Disadvantage: changes our view of the program from what it means to how it’s executed.

Allocating space on the stack - 1

Each time a query or a subgoal is executed, a “stack frame” is added to the stack.

If the subgoal is a recursive call and the recursion is deep, then there will be many stack frames added – potentially wasting much space.
Allocating space on the stack - 2

A growing stack

- Frame n
- Frame n-1
- Frame n-2

A static-size stack

- Frame n
- Frame n-1
- Frame n-2

How can stack growth be restricted?

Last call optimization (LCO) and accumulators - 1

LCO is a space optimization technique that is automatically applied at the point when the last subgoal in a rule is to be called when a procedure is deterministic.

This is often used with accumulators - an accumulating parameter builds up a result as the computation proceeds.

Last call optimization (LCO) and indexing - 1

Consider the problem of checking whether a given vowel is an "a" and updating a counter. A straightforward version might be:

% 1 - terminating
update_count(a, A, A1) :-
A1 is A + 1.

% 2 - not "a"
update_count(Letter, A, A) :-
Letter \= a.

However, this would waste much space on the stack.

Adding an accumulator - 1

Adding an accumulator involves adding another argument:

% 1 - terminating
fac2(0, Fact, Fact).
% 2 - recursive
fac2(N, Accum, Fact) :-
N1 is N - 1,
tab(N), write(Accum), nl,
Accum1 is N * Accum,
fac2(N1, Accum1, Fact).

This isn't tail recursive and so grows the stack, so we need to rewrite it.

Adding an accumulator - 2

Why are accumulators of importance in Prolog?

Because they are an important way of ensuring that non-deterministic subgoals are called last and so enabling last call optimization.
Prolog's execution strategy explained more precisely

Use of if … then … else - 1

Prolog expresses “or” by having alternative clauses in a procedure - and this is usually the best method. But, there are significant efficiency advantages when the first subgoals of rules are based on:

– arithmetic comparisons;
– type tests.

We can use Prolog’s if...then...else and the compiler produces very efficient code.

A less space-wasteful version is:

\% 1 - not "a"
update_count(Letter, A, A) :-
    Letter \== a.
\% 2 - a
update_count(a, A, A1) :-
    A1 is A + 1.

Use of if … then … else - 2

Example based on factorial:

fac3(N, Acc, Fact) :-
    ( N > 0 ->
        N1 is N - 1,
        Acc1 is N * Acc,
        fac3(N1, Acc1, Fact)
    ;
    N =:= 0 ->
        Acc = Fact
    ).

Cuts

Prolog includes a predicate called “cut” and written as “!” which allows the stack to be manipulated during the running of a program.

This is bad thing!

Cuts are classified into two patterns of use:

– green cuts
– red cuts

Green cuts

Don’t change the “meaning” of the program but stops Prolog storing unnecessary backtracking information on the stack.

Take away the cut and the program will produce the same results.

Red cuts

Change the “meaning” of the program.

Take away the cut and the program will produce the different results.

Some people believe that programmers need two years of experience before using the cut.
Consider the following code:

```prolog
max1(X, Y, X) :-
    X >= Y, !. % green cut
max1(X, Y, Y) :-
    X < Y.
```

Clearly this computes the maximum of two numbers - and it is correct.

The test in the second clause should be superfluous - if the first number isn’t smaller than the first, the second clause applies:

```prolog
max2(X, Y, X) :-
    X >= Y, !. % red cut
max2(_, Y, Y).
```

Conclusion: *cuts need a lot of care.*

---

**Summary - 1**

A simplistic model Prolog’s execution method hides several optimizations which can be quite powerful.

Indexing is the simplest, very powerful and helps us design the order of arguments.

Last call optimization suggests ways in which clauses in a procedure should be ordered.

**Summary - 2**

Last call optimization is powerful but often requires the use of accumulators.

The *if…then…else* construct can be efficient in very limited circumstances.

Cuts are difficult to apply and can lead the inexperienced into great difficulties.