Worksheet 15

Talking to the Prolog Interpreter

The previous two worksheets showed extensions to Prolog that offered extralogical versions of logical things. This worksheet introduces decidely non-logical things. Prolog offers evaluable predicates that allow the programmer to ask about and influence the state of the computation. The aims of this worksheet are

- that you see the main extralogical facility Prolog provides for controlling backtracking;
- that you should see the consequences that this provision has on declarativeness and correctness; and
- that you see facilities for asking about the state of the computation.

15.1 Introduction

There is no special emphasis in this course on writing efficient Prolog. I have tried to keep the code clear, rather than make it efficient. The point of view implicit in this is that it is easier to make a correct program more efficient than an incorrect program more correct.

Efficiency in Prolog is as much about good choices of algorithms and data structures as it is in any other language. But even when we've chosen these carefully, we may require yet further efficiency (particularly given that in Prolog we surrender so much control over the execution to the system software). While we could consider very system specific aspects of efficiency (e.g. how your implementation of Prolog indexes the clauses in the database) and how we might exploit these in clever tricks, or the advantages that might accrue from using a compiler (particularly an optimising compiler), we won't concern ourselves with these. There is, after all, a much more obvious aspect of Prolog that we should look at first: its search.

Prolog's automatic backtracking gives it its power and is part of what allows us to specify the 'what' and not the 'how'. For example, in permutation sort,

```
permsort(OriginalList, SortedList) :-
   perm(OriginalList, SortedList),
   is_ordered(SortedList).
```

Prolog generates a permutation of the original list, tests to see whether it is ordered, and backtracks if it is not; on backtracking, it generates another permutation, tests this, and so on until it does generate one that is ordered.[1]

While this inbuilt search is a source of power and declarativeness, it is also a source of inefficiency, particularly when Prolog tries fruitless branches. Sometimes, for the sake of efficiency, you might want to prevent backtracking by pruning fruitless branches from the search tree. Prolog provides the cut operator for this purpose.

15.2 The Cut

15.2.1 What it does

The cut has no declarative semantics; it is an operator for affecting the procedural behaviour of programs. It reduces the search space by dynamically pruning the search tree. It can be used to prevent Prolog from following paths that the programmer knows could not produce solutions. Of course, it can, therefore, also be used (inappropriately) to prune paths that do contain solutions.

The cut is written as a subgoal in the body of a Prolog rule as an exclamation mark. Operationally, when Prolog encounters the cut in the body of a rule, the cut succeeds and commits Prolog to all the choices made since the parent goal was unified with the head of the clause the cut occurs in. Prolog can still try to get several solutions to the goals to the right of the cut using backtracking. But once these have been found, there is no backtracking beyond the cut. Alternative solutions to the goals to the left of the cut are pruned; alternative clauses that also match the parent goal but appear *subsequent* to the current clause are pruned.

More schematically, consider a goal `Goal` which matches with the head `Head` of the following clause:

[1] Programs that embody this problem-solving method are even more common in A.I. problems, where they are referred to as *generate-and-test* programs.

```
Head :- SubGoal1,...,SubGoali, !, SubGoalj,...,SubGoaln.
```

Assume that the parent goal, `Goal`, has matched with `Head` and that a solution to `SubGoal1,...SubGoali` has been found. When the cut (!) is encountered, this solution to `SubGoal1,...,SubGoali` becomes frozen. The system can go on to solve `SubGoalj,...,SubGoaln` in as many ways as possible but it cannot backtrack to find alternative solutions to `SubGoal1,...SubGoali`: all alternatives are discarded. Furthermore, the goal `Goal` now becomes committed to this clause: no attempt is made to match `Goal` with subsequent clauses in the database.

Informally, the cut says: 'once you've reached me, stick to all the variable substitutions you've found after you entered my clause; don't try to find any alternatives to my left, and don't try any alternative subsequent clauses. (Don't you just love anthropomorphism?)

□ Try the following example out for yourself.

Enter the following facts:

```
a(25).
b(26).
c(1).
c(2).
c(3).
d(2).
d(3).
e(1).
e(2).
f(1).
f(2).
g(10).
h(12).
```

and the following rules (which have no cuts in them):

```
i(X, Y) :- a(X), b(Y).
i(X, Y) :- c(X), d(X), e(Y), f(Y).
i(X, Y) :- g(X), h(Y).
```

Here are the solutions to a simple query:

```
?- i(X, Y).

Y = 26
X = 25 ;
```

```
Y = 1
X = 2 ;

Y = 2
X = 2 ;

Y = 1
X = 3 ;

Y = 2
X = 3 ;

Y = 12
X = 10 ;

no
```

But now change the second of the clauses that define `i` to:

```
i(X, Y) :- c(X), d(X), !, e(Y), f(Y).
```

Now the query produces fewer solutions:

```
?- i(X, Y).

Y = 26
X = 25 ;

Y = 1
X = 2 ;

Y = 2
X = 2 ;

no
```

The first clause that defines `i` works as before (giving the first solution). The second clause, on finding `X = 2` becomes, by virtue of the cut, committed to this binding; all solutions to `e(Y), f(Y)` are found but there is no backtracking to find `X = 3` as a solution to `c(X), d(X)`; nor does the system try the third clause that defines `i`.

Add the following to the program:

```
j(X, Y) :- i(X, Y).
j(X, Y) :- k(X, Y).
```

```
k(51, 52).
k(53, 54).
```

With the cut still in the second clause that defines `i`, consider the following:

```
?- j(X, Y).

Y = 26
X = 25 ;

Y = 1
X = 2 ;

Y = 2
X = 2 ;

Y = 52
X = 51 ;

Y = 54
X = 53 ;

no
```

You will note that the cut affects the solutions of `i` that are found, but does not affect choice points 'above' the clause containing the cut, i.e. clauses calling `i`, in this case `j`.

We now consider how cuts might be deployed.

15.2.2 Green Cuts

Consider, the following program which computes the minimum of two numbers:

```
min(X, Y, X) :- X =< Y.
min(X, Y, Y) :- X > Y.
```

We know that the two clauses are mutually exclusive: if the first clause succeeds, the second will fail. (Of course, *we* know this, but *Prolog* can't be expected to work it out.)

We can prevent Prolog from backtracking and trying the second clause if the first succeeds by using the cut operator, thus:

```
min(X, Y, X) :- X =< Y, !.
min(X, Y, Y) :- X > Y.
```

Here is a trace of the cut-free version:

```
?- min(2, 3, M).

( 1) 1 call:  min(2,3,_0) ?
( 2) 2 call:  2=<3 ?
( 2) 2 exit:  2=<3
( 1) 1 exit:  min(2,3,2)

M = 2 ;

( 2) 2 back:  2=<3 ?
( 2) 2 fail:  2=<3
( 3) 2 call:  2>3 ?
( 3) 2 fail:  2>3
( 1) 1 fail:  min(2,3,_0)

no
```

And here is a trace of the version with the cut:

```
?- min(2, 3, M).

( 1) 1 call:  min(2,3,_0) ?
( 2) 2 call:  2=<3 ?
( 2) 2 exit:  2=<3
( 1) 1 exit:  min(2,3,2)

M = 2  ;

no
```

In the first version, when I type semi-colon to ask for further answers, the program tries the other clause, which fails, so this is pointless effort. In the second version, when I type semi-colon, having reached the cut, no backtracking is allowed and so failure is immediate.

The new program, equipped with cuts, is in general more efficient than the original version without cuts. The new program 'recognises' that there are no further solutions sooner than the original program does. We have been able to exploit the fact that `min` is really a deterministic program by making its execution deterministic.

This use of the cut is entirely 'innocent'. It has no effect on the declarative meaning of the program (it merely improves efficiency). If the cuts are now removed, the program will still produce the same result; it will perhaps only spend more time.

Equally, and this merely emphasises that the declarative meaning is unchanged, we could swap the order of the clauses (even leaving the cut in) and the program will still work (but more slowly) (provided, as here, there are no problems with left-recursion, arithmetic argument instantiation or negation argument instantiation). People reading the program can ignore them. It does not affect their understanding. Someone proving things about the program can ignore them. Such innocent uses of cut are termed *green cuts*. A cut is green if it does not cut away success branches.

15.2.3 Red Cuts

There is another source of inefficiency in the `min` program.

We know that if the first clause's test fails then the second clause's test is bound to succeed. We conclude that the second clause's test is redundant and can be omitted:

```
min(X, Y, X) :- X =< Y, !.
min(X, Y, Y).
```

This looks even more efficient because it saves on an unnecessary looking test. You can see this by looking at its behaviour on the goal `?- min(3, 2, M)`.

However, this use of cut very much exploits the sequential processing of rules in Prolog, and is a misguided style of programming.

The first obvious consequence is that, if you remove the cut, the program may produce multiple solutions, some of which are not correct. (For example, without this cut, this program would give both *M = 2* and *M = 3* as answers to the query `?- min(2, 3, M)`.) So the procedural meaning and the declarative meaning of the program have diverged.

What's more, even the *procedural* meaning of this program is wrong (i.e. it's not what you want and not what you'd expect). As an illustration of this, consider the goal `?- min(1, 2, 2)`. i.e. we are asking Prolog to verify that the minimum of 1 and 2 is 2. In case you don't realise: this is wrong and ought to get the answer *no*. However, with this version of this program we get the answer *yes*:

```
?- min(1, 2, 2).
yes
```

The goal does not unify with the head of the first clause. But it does unify with the head of the second clause; this clause has no body, so we get an immediate answer *yes*. Therefore it is quite common for these badly used cuts to give wrong answers or to change the meaning.

Other consequences include: swapping the order of the clauses is no longer permissible as it was with green cuts; and the program is harder to understand because the reader has to understand what the cuts are doing. Such nasty cuts are called *red cuts.* A cut is red if it cuts away success branches.

15.3 Exercises

Exercise 15.1 *In this program*

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y) :- X < Y.
```

is the cut red or green? Change the program so that the cut changes colour. For the version with the red cut, find a goal on which the program is wrong.

Also, compare the version above, with the following:

```
max(X, Y, X) :- !, X >= Y.
max(X, Y, Y) :- X < Y.
```

What is the difference between this version, the original version and the version you came up with?

Exercise 15.2 *The cut in the following is red:*

```
member(X, [X|Xs]) :- !.
member(X, [Y|Ys]) :- member(X, Ys).
```

What has changed in the meaning of this version compared with the cut-free version?

Exercise 15.3 *Define* `split(Numbers, Positives, Negatives)` *which splits a list of numbers into two lists: positive ones including zero, and negative ones. Give three versions: one that is cut free, one that has a green cut and one that has a red cut. Compare their efficiency by tracing.*

Exercise 15.4 *Here is a surprising example of a red cut: the definition of negation as failure:*

```
nff(P) :- call(P), !, fail.
nff(P).
```

This program also uses the inbuilt primitive `call` *(for a discussion of which, see the previous worksheet).*

The program takes `P` *and tries to prove it, using* `call`*. If this succeeds, the cut is encountered, preventing backtracking, and the goal* `fail` *is encountered,* *which obviously fails.*[2] *And so* `no` *is returned. This is right:* `P` *was successful, so* `no` *is the answer to its negation. Alternatively, if* `P` *failed, the second clause would succeed trivially and* `yes` *would be returned.*

The exercise is to define a similar predicate that mimics the operation of conditional expressions in imperative languages: `if_then_else(P, Q, R)` *should mimic* `if P then Q else R`*. (Do not ever use this in any other part of this course.*[3]*)*

Cut is a bit like GOTO in imperative languages. You clean up the dangers of using GOTOs by wrapping their use up in nice looping and conditional constructs. Similarly, good Prolog programming style is to avoid direct use of cuts, but to use negation and conditionals to wrap the cuts inside well-behaved constructs.

15.4 Asking about the state of the computation

The main evaluable predicate for asking about (one aspect of) the state of the computation is `var/1` (and its reverse `nonvar/1`). `var/1` tests whether its argument is at present an uninstantiated variable. For example, if we have the following fact in the database:

```
male(alf).
```

then here are uses of `var/1`:

```
?- var(X).
yes

?- male(X), var(X).
no

?- var([X|Xs]).
no
```

`var/1` and `nonvar/1` are useful in metaprogramming (worksheet 16), though we'll avoid it. I used it in my implementation of a unification program (look in the file `unify`), where we clearly need to know whether something is a variable or not.

[2] In some implementations of Prolog, `fail` is a reserved word and is defined as a goal that always fails. It needn't be: all you need to do is to make sure you give no definition of `fail` in your program.

[3] Many Prologs come with something along these lines built in; it can be a very efficient way of dealing with deterministic conditionals, but I wish to discourage its use in this course, otherwise you'll all write imperative programs in Prolog.

And, we can use it to trap cases where queries use modes that cause a program some trouble. For example, if you'd rather `member/2` didn't ever launch an infinite computation then you can make sure that its second argument is instantiated:

```
member(X, Ys) :- nonvar(Ys), Ys = [X|Xs].
member(X, Ys) :- nonvar(Ys), Ys = [Z|Zs], member(X, Zs).
```

Or you can restore polymodality while using system predicates that would otherwise give error messages if given uninstantiated arguments, e.g.

```
polylength(Xs, N) :- nonvar(Xs, length1(xs, N).
polylength(Xs, N) :- var(Xs), nonvar(N), length2(Xs, N).

length1([], 0).
length1([X|Xs]) :- lenght1(Xs, N1), N is N1 + 1.

length2([], 0).
length2([X|Xs], N) :- N > 0, N1 is N - 1, length2(Xs, N1).
```

(of course, this isn't in the spirit of polymodality.)

☐ Try the above out.

Acknowledgements

Some material is based on S&S and Bratko.