The 'cut' and other control primitives

Marek Sergot
Department of Computing
Imperial College, London

Autumn 2013

1 Example

```
entitled_to(X, Payment) :-
   below_poverty_limit(X),
   poor_rate(Payment).
entitled_to(X, Payment) :-
   \+ below_poverty_limit(X),
   ordinary_rate(Payment).
```

Consider the computational behaviour of the query

```
?- entitled_to(peter, Payment).
```

Suppose that `below_poverty_limit(peter)` involves a very lengthy and expensive computation.

Case 1 Suppose `peter` is not below the poverty limit.

- Prolog tries the first `entitled_to` clause first.
- The first condition `below_poverty_limit(peter)` fails eventually after a long computation in which all ways of showing `below_poverty_limit(peter)` are tried.
- Prolog now tries the second `entitled_to` clause.
- The first condition `\+ below_poverty_limit(peter)` succeeds if and only if all ways of showing `below_poverty_limit(peter)` fail.
- They will — we just did *exactly* this computation.

Case 2 Suppose `peter` is below the poverty limit.

- Prolog tries the first `entitled_to` clause first.
- The first condition `below_poverty_limit(peter)` succeeds eventually and we get an answer.
- If we ask for another answer, or do say `findall` to find them all, Prolog backtracks:
  (a) to find another way of showing `below_poverty_limit(peter)`
  (even though we already know this). And then:
  (b) to try the second `entitled_to` clause.
  This second clause is bound to fail, but Prolog will try it anyway.

Note There are still some inefficiencies in this second case, even if we tell Prolog (somehow) to stop after finding one solution. Because there is the *possibility* that the second `entitled_to` clause will be required, Prolog will set up the appropriate backtracking points in anticipation. Backtracking points use up computer memory.

2 Example

```
pensioner(X) :-
   male(X),
   age(X, Age),
   Age >= 65.
pensioner(X) :-
   female(X),
   age(X, Age),
   Age >= 60.

male(peter).                age(peter, 64).
male(frank).                age(frank, 66).
male(colin).                age(colin, 67).
```

(Suppose there are many – millions and millions – of facts about `male`, `female`, and `age`.)

Consider the query

```
?- pensioner(peter).
```

Prolog tries the first clause for `pensioner`. The `male(X)` condition succeeds with `X = peter`; the condition `age(peter, Age)` succeeds with `Age = 64`. The last condition `64 >= 65` fails.

Now Prolog backtracks, and it backtracks to continue looking for more solutions to

```
age(peter, Age)
```

There aren't any, but Prolog has no way of knowing that persons only have one age.

(There is a little point of detail here. In fact the Sicstus compiler, and several other Prolog compilers, indexes all clauses on the first argument of the head. If this is not a variable then the Sicstus compiler can reduce the search to simple look-ups using its indexing.)

3 Example

```
happy(X) :- person(X), likes(_, X).

likes(jim, frank).
likes(chris, frank).
likes(dave, chris).

person(jim).
person(chris).
person(dave).
person(frank).
```

Suppose we ask for all solutions to the query

```
?- happy(X).
```

Prolog gives the answers:

```
X = chris
X = frank      % because jim likes frank
X = frank      % because chris likes frank
```

Note: Once again, the backtracking points have to be set up here even if only one answer to the query is requested.

4 The 'cut'

Syntactically the cut (`!`) can be written as an extra condition anywhere in the bodies of rules or queries.

Procedurally it means, when executed:

- remove all current backtracking points

In practice it means, for example in

$$A \texttt{ :- } B\texttt{, !, } C$$

that if B succeeds and the cut `!` is executed:

- do not try any other clauses for A — i.e., *commit* to using this clause only;
- do not try finding any other solutions to earlier goals (here, B).

4.1 The pensioner example with cuts

```
pensioner(X) :-
   male(X),
   age(X, Age), !,
   Age >= 65.
pensioner(X) :-
   female(X),
   age(X, Age), !,
   Age >= 60.
```

The presence of the 'cut' means that Prolog will not backtrack for other solutions to `age(X, Age)`.

It also means, however, that Prolog will not backtrack for other solutions to `male(X)` and `female(X)`.

Example

```
male(peter).                    age(peter, 64).
male(frank).                    age(frank, 66).
male(colin).                    age(colin, 67).


?- pensioner(frank).
yes

?- pensioner(colin).
yes

?- pensioner(X).
X = frank ;     % backtrack
no              % what happened to colin?
```

4.2 The entitlement example with cuts

```
entitled_to(X, Payment) :-
   below_poverty_limit(X),
   poor_rate(Payment).
entitled_to(X, Payment) :-
   \+ below_poverty_limit(X),
   ordinary_rate(Payment).
```

The redundant re-computation of `below_poverty_limit(X)` can be eliminated by inserting a 'cut':

```
entitled_to(X, Payment) :-
   below_poverty_limit(X), !,
   poor_rate(Payment).
entitled_to(X, Payment) :-
   ordinary_rate(Payment).
```

The modified program works for the input-output pattern

given `X`, find `Payment`

But notice:

- the declarative (logical) reading of the program has been destroyed;
- the order of the clauses is significant (not just for efficiency, but for answers computed);
- the second clause, read in isolation is *not true*!

5 Problems with the cut

The cut — nearly always — destroys the declarative reading of a program.

This means that the program may not behave correctly when used in different, unanticipated ways.

Example

Adam and Eve have no parents. Everyone else has two.

```
number_of_parents(adam, 0) :- !.
number_of_parents(eve, 0) :- !.
number_of_parents(X, 2) :- person(X).

person(adam).
person(eve).
person(jim).
```

Now consider the following queries:

```
?- number_of_parents(eve, N).
N = 0      % no other solutions

?- number_of_parents(jim, N).
N = 2      % no other solutions

?- number_of_parents(eve, 2).
yes      % !!!

?- number_of_parents(X, 0).
X= adam ;     % backtrack
no            % what happened to eve??

?- number_of_parents(X, N).
X= adam, N = 0 ;     % backtrack
no
```

Solution 1 (partial)

```
number_of_parents(adam, N) :- !, N = 0.
number_of_parents(eve, N) :- !, N = 0.
number_of_parents(X, 2) :- person(X).
```

Solution 2

```
number_of_parents(adam, 0).
number_of_parents(eve, 0).
number_of_parents(X, 2) :-
    person(X),
    X \= adam, X \= eve.
```

Solution 2 is declaratively correct. (And calls to `=` are very quick.)

6 Alternatives to the cut

Prolog provides a number of built-in control primitives of which the most important and useful is the 'if-then-else' construct

$$(P \texttt{ -> } Q \texttt{ ; } R) \quad \text{— if } P \text{ then do } Q \text{ else do } R$$

P, Q, R can be atoms or conjunctions of atoms, i.e., of the same form as queries and bodies of rules.

From the Sicstus manual:

> (P `->` Q `;` R) is defined as if by
> (P `->` Q`;` R) `:-` P`, !,` Q`.`
> (P `->` Q`;` R) `:-` R`.`
> except the scope of any cut in Q or R extends beyond the if-then-else construct.
> In sicstus execution mode no cuts are allowed in P. In iso execution mode cuts are allowed in P and their scope is the goal P.

I have no idea what is meant by the part 'except the scope of any cut in Q or R ...'. I have never, ever written a program with cuts in any of P, Q, R above, and I can't imagine why I would ever want to.

The last bit from the manual is much more important:

> Note that this form of if-then-else only explores the first solution to the goal P.

This is very important.

6.1 The entitlement example

```
entitled_to(X, Payment) :-
   (below_poverty_limit(X)
    ->
       poor_rate(Payment)
    ;
       ordinary_rate(Payment)
   ).
```

(The white space, newline, indentation are not significant. They are for readability.)

This will produce at most one solution for given `X` even if there are many ways of solving `below_poverty_limit(X)` and many solutions to `poor_rate(Payment)` and `ordinary_rate(Payment)`.

- If `below_poverty_limit(X)` succeeds and `poor_rate(Payment)` fails then `entitled_to(X, Payment)` fails.
- If `below_poverty_limit(X)` fails and `ordinary_rate(Payment)` fails then `entitled_to(X, Payment)` fails.

6.2 The pensioner example

```
pensioner(X) :-
   age(X, Age),
   (male(X)
    ->
       Age >= 65
    ;
       Age >= 60
   ).
```

The above will backtrack to `age(X, Age)` even for `X` given.

One *could* also write it like this. (This is just for illustration. Some might look strange, but see next example.)

```
pensioner(X) :-
   male(X),
   (age(X, Age)
    ->
       Age >= 65 ; fail
    ).
pensioner(X) :-
   female(X),
   (age(X, Age)
    ->
       Age >= 60 ; fail
    ).
```

Or like this:

```
pensioner(X) :-
   male(X),
   (age(X, Age), Age >= 65
    ->
       true ; fail
    ).
pensioner(X) :-
   female(X),
   (age(X, Age), Age >= 60
    ->
       true ; fail
    ).
```

(Or in several other ways.)

6.3 Example

```
happy(X) :- person(X), likes(_, X).
```

gives many copies of `X` if there are many persons who like `X`.

Recall that `(`P `->` Q `;` R`)` only explores the *first* solution to the goal P.

So:

```
happy(X) :- person(X),
            (likes(_, X) -> true ; fail).
```

Read it procedurally!!

If you just want to *test* `likes(_, X)` you could also write

```
happy(X) :- person(X),
            \+ \+ likes(_, X).
```

7 Some other constructs

`(`P `->` Q`)`	–	same as `(`P `->` Q `; fail)`
`once(`P`)`	—	same as `(`P `-> true)`

So the previous example could be written:

```
happy(X) :- person(X),
            (likes(_, X) -> true).
```

or

```
happy(X) :- person(X),
            once(likes(_, X)).
```

Important note Do *NOT* read `(`P `->` Q`)` as the logic expression $P \to Q$. $P \to Q$ is true if P is false. `(`P `->` Q`)` fails if P fails. Read the 'then' in `(`P `->` Q`)` as 'P and then Q' or 'if P then do Q'.

(Personally I never use `(`P `->` Q`)` or `once(`P`)` but only the `(`P `->` Q `;` R`)` 'if-then-else' form in full. Declaratively, you can read `(`P `->` Q `;` R`)` as $(P \wedge Q) \vee (\neg P \wedge R)$. But usually it is the procedural reading that will make sense.)

There are some other control constructs in Prolog. See the manual. (I never use them.)

7.1 An example with nested 'if then else'

The 'if then else' constructs can be nested, like this, to give a kind of 'case statement'. (The white space, newlines, indentations, are just for readability.)

The following is from the Department of Computing's Grading Scheme for assessed courseworks:

```
grade(Mark, Grade) :-
   integer(Mark),           % fails if Mark is not an integer
   0 =< Mark, Mark =< 100,  % between 0 and 100
   (  Mark >= 90 ->
        Grade = 'A+'
    ; Mark >= 70, Mark < 90 ->
        Grade = 'A'
    ; Mark >= 60, Mark < 70 ->
        Grade = 'B'
    ; Mark >= 50, Mark < 60 ->
        Grade = 'C'
    ; Mark >= 40, Mark < 50 ->
        Grade = 'D'
    ; Mark >= 30, Mark < 40 ->
        Grade = 'E'
    ; % otherwise
        Grade = 'F'
   ).
```

For this example, the above can also be written equivalently as:

```
grade(Mark, Grade) :-
   integer(Mark),           % fails if Mark is not an integer
   0 =< Mark, Mark =< 100,  % between 0 and 100
   (  Mark >= 90 ->
        Grade = 'A+'
    ; Mark >= 70 ->
        Grade = 'A'
    ; Mark >= 60 ->
        Grade = 'B'
    ; Mark >= 50 ->
        Grade = 'C'
    ; Mark >= 40 ->
        Grade = 'D'
    ; Mark >= 30 ->
        Grade = 'E'
    ; % otherwise
        Grade = 'F'
   ).
```


8 Final remarks

Here are some examples of *very bad* programming. (They are from actual MSc student submissions in previous years.)

8.1 Awful example 1

The task was to define a predicate `cls(X)` which holds when `X` represents a clause. A *literal* is by definition either a logical atom, or the negation of a logical atom; a *clause* is a literal or a disjunction of literals.

Here is one solution in Prolog:

```
cls(X) :- logical_atom(X).
cls(neg(X)) :- logical_atom(X).
cls(or(X,Y)) :- cls(X), cls(Y).
```

The program for `logical_atom` was provided. It doesn't matter for this example. (There is no need for cuts here as `logical_atom(X)` fails if `X` is a term of the form `neg(_)` or `or(_,_)`.)

Here is the kind of *very bad* program which some people seem determined to write:

```
cls(X) :- logical_atom(X), !.
cls(neg(neg(X)) :- !, fail.
cls(neg(X)) :- cls(X).
cls(or(X,Y)) :- cls(X),cls(Y).
cls(and(X,Y)) :- !,fail.
cls(imp(X,Y)) :- !,fail.
```

Apart from being wrong, this program is completely obscure. And the last two clauses serve *no purpose at all.*

8.2 Awful example 2

One year someone wrote a program containing the following clause:

```
cls(F) :- logical_atom(F) -> (!,true);(fail,!).
```

I have *absolutely no idea* what this is supposed to mean. I can make no sense of it at all. It is surprising how many people write this kind of thing.