Chapter 5

Controlling Backtracking:
Chapter 5
Controlling Backtracking

- Preventing backtracking
- Examples using cut
- Negation as failure
- Problems with cut and negation
5.1 Preventing backtracking

- Prolog will automatically backtrack if this is necessary for satisfying a goal.

- Uncontrolled backtracking may cause inefficiency in a program

- ‘Cut’ can be used to control or prevent backtracking
Experiment 1

Double step function

- Rule 1: if $X < 3$ then $Y = 0$
- Rule 2: if $3 \leq X$ and $X < 6$ then $Y = 2$
- Rule 3: if $6 \leq X$ then $Y = 4$

In Prolog,

```prolog
f (X,0) :- X < 3.          % Rule 1
f (X,2) :- 3 =< X, X < 6.   % Rule 2
f (X,4) :- 6 =< X.          % Rule 3
```
Experiment 1 continued

Question:

?- f(1,Y), 2 < Y.

The first goal f(1,Y) instantiated Y to 0. The second goal becomes 2 < 0 which fails.

Prolog tries through backtracking two useless alternatives (Rule 2 and Rule 3)
Experiment 1 continued

- The three rules are mutually exclusive and one of them at most will succeed
- as soon as one of them succeeds there is no point in trying to use the others as they are bound to fail
- ‘cut’ is used in this case
Experiment 1 continued

\[ f (X,0) : X < 3, !. \] % Rule 1
\[ f (X,2) : 3 \leq X, X < 6, !. \] % Rule 2
\[ f (X,4) : 6 \leq X. \] % Rule 3

?- f (1,Y), 2 < Y.

Prolog choose rule 1 since \( 1 < 3 \) and fails the goal \( 2 < Y \) fails. Prolog will try to backtrack, but not beyond the point marked ! in the program. Rule 2 and rule 3 will not be generated.
**Experiment 2**

```
f (X,0) :- X < 3, !.                        % Rule 1
f (X,2) :- 3 =< X, X < 6, !.              % Rule 2
f (X,4) :- 6 =< X.                            % Rule 3

?- f (7,Y).
Y = 4

- try rule 1: 7 < 3 fail, backtrack, try rule 2
- try rule 2: 3 =< 7 succeeds, 7 < 6 fails, backtrack and try rule 3
- try rule 3: 6 =< 7 succeeds
- not efficient
```
more economical formulation of the three rules with cut:

- if $X < 3$ then $Y = 0$,
- otherwise if $X < 6$ then $Y = 2$,
- otherwise $Y = 4$.

$f (X,0) :- X < 3, !.$ % Rule 1
$f (X,2) :- X < 6, !.$ % Rule 2
$f (X,4).$ % Rule 3
Affecting procedural behavior without cut

\[ f(X,0) : - X < 3. \quad \text{% Rule 1} \]
\[ f(X,2) : - X < 6. \quad \text{% Rule 2} \]
\[ f(X,4). \quad \text{% Rule 3} \]

producing multiple solutions

?- f (1,Y).

Y=0
\text{Y=2}
\text{Y=4}
\text{no}
Chapter 5
Controlling Backtracking

- Preventing backtracking
- Examples using cut
- Negation as failure
- Problems with cut and negation
5.2 Examples using cut

- Finding maximum

max(X, Y, Max)

where

max = X if X >= Y and
max = Y if X < Y

In Prolog,

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

They are mutually exclusive. Only one can succeed.
More economical formulation rules with cut:

\[ \text{max}(X,Y,\text{Max}) \]
where
\[
\begin{align*}
\text{If } X &\geq Y \text{ then } \text{Max} = X \\
\text{otherwise } \text{Max} &\mathbf{= Y}
\end{align*}
\]

In Prolog,
\[
\begin{align*}
\text{max}(X,Y,X) &\leftarrow X \geq Y, !. \\
\text{max}(X,Y,Y) &. 
\end{align*}
\]
Handle with care

?- max(3,1,1).
yes
Actually it should be no since 3 > 1.

- reformulation to overcome this limitation

max(X,Y,Max) :- X >= Y, !, Max = X ;
               Max = Y.
Single-solution membership

- Non-deterministic: if X occurs several times
  
  \[
  \text{member}(X,[X|L]). \\
  \text{member}(X,[Y|L]) \leftarrow \text{member}(X,L).
  \]

- Deterministic
  
  \[
  \text{member}(X,[X|L]) \leftarrow !. \\
  \text{member}(X,[Y|L]) \leftarrow \text{member}(X,L).
  \]
Adding an element to a list without duplication

\[ \text{add}(X,L,L1) :- L1 = [X|L] . \]

If \( X \) is a member of list then \( L1 = L \)
otherwise \( L1 \) is equal to \( L \) with inserted

\[ \text{add}(X,L,L) :- \text{member}(X,L),! . \]
\[ \text{add}(X,L,[X|L]). \]
Classification into categories

beat(tom,jim).
beat(ann,tom).
beat(pat,jim).

class(X,fighter) :- beat(X,_),
                beat(_,X),!.

class(X,winner) :- beat(X,_), !.

class(X,sportsman) :- beat(_,X).
Handle with care

?- class(tom,C).
C = fighter
no

?- class(tom,sportsman).
Yes

% as intended
% not as intended
Chapter 5
Controlling Backtracking

- Preventing backtracking
- Examples using cut
- Negation as failure
- Problems with cut and negation
5.3 Negation as failure

- Mary likes all animals but snakes
- How can we write this in Prolog?
- Mary likes any X if X is an animal
- In Prolog,
  
  
  likes(mary, X) :- animal(X) .
How can we exclude snakes?

If X is a snake then ‘Mary likes X ‘ is not true, otherwise if X is an animal then Mary likes X.

In Prolog,
likes(mary, X) :-
    snake(X), !, fail.
likes(mary, X) :-
    animal(X).
Using **different** relation

different(X,Y)

which is true if X and Y are different

different can be understood in several ways:
- X and Y are not literally the same
- X and Y do not match
- the values of arithmetic expression X and Y are not equal
Using \textbf{different} relation in Prolog

- If X and Y match then \texttt{different}(X,Y) fails,
- otherwise \texttt{different}(X,Y) succeeds.

\texttt{different}(X,X) :- !, fail.
\texttt{different}(X,Y).

Another way to write only in one clause

\texttt{different}(X,Y) :-
    X = Y, !, fail
    ;
    true.  \% true is goal that always succeeds
‘not’ relation

- not(Goal) is true if Goal is not true
- If Goal succeeds then not(Goal) fails,
- otherwise not(Goal) succeeds

- In Prolog,
  
  ```prolog
  not(P) :- P, !, fail ; true.
  ```
as a prefix operator

not (snake(X))
as:
not snake(X)

- In snake problem

likes(mary, X) :-
  animal(X),
  not snake(X).
Chapter 5
Controlling Backtracking

- Preventing backtracking
- Examples using cut
- Negation as failure
- Problems with cut and negation
5.4 Problems with cut and negation

Advantages for using cut

- improve the efficiency of the program
- able to specify mutually exclusive rules
  - if condition P then conclusion Q,
  - otherwise conclusion R

Disadvantages

- lose the valuable correspondence between the declarative and the procedural meaning
- change of order of clauses may affect the declarative meaning
declarative meaning

Green cuts
- no effect on declarative meaning

Red cuts
- do affect the declarative meaning

\[ p : - a, l, b. \]
\[ p : - c. \]

declarative meaning is \( p \iff (a\&b) \lor (\neg a\&c) \)

\[ p : - c. \]
\[ p : - a, l, b. \]

declarative meaning is \( p \iff c \lor (a\&b) \)
Negation ‘not’

- When processing a not goal, Prolog does not try to prove this goal directly.
- It tries to prove the opposite, and if the opposite cannot be proved then Prolog assumes that the not goal succeeds.

Closed world assumption

- everything that exists is stated in the program or can be derived from the program
- if something is not in the program or cannot derived from it then it is not true and consequently its negation is true
Restaurant example

good_standard(jeanluis).
expensive(jeanluis).
good_standard(francesco).
reasonable(Restaurant) :-
    not expensive(Restaurant).

?- good_standard(X), reasonable(X).
X = francesco

?- reasonable(X), good_standard(X).
no
Why?

In the previous example

- Variable $X$ is, in the first case, already instantiated when `reasonable(X)` is executed
- whereas $X$ is not yet instantiated in the second case

In general

- `not Goal` works safely if the variables in `Goal` are instantiated at the time `not Goal` is called
- otherwise we may get unexpected results due to reasons explained in the sequel
Quantification to Universal

?- expensive(X).
- means: Does there exist X such that expensive(X) is true?
- If yes, what is X? So X is existentially quantified. Accordingly, Prolog answer X=jeanluis

?- not expensive(X).
- Prolog does not interpret as: Does there exist X such that not expensive(X) ?
- Prolog interprets as: not (exists X such that expensive(X) )
- That is, For all X : not expensive(X)
Summary

- cut prevents backtracking. It improves the efficiency of the programs and enhance the expressive power of the language.

- cut makes it possible to formulate mutually exclusive conclusion through rules of the form:
  - if Condition then Conclusion 1 otherwise Conclusion 2

- cut may affect the declarative meaning, and need to use with care.

- not defined through failure does not exactly correspond to negation in mathematical logic. The use of not also requires special care.