5 Controlling Backtracking

We have already seen that a programmer can control the execution of a program through the ordering of clauses and goals. In this chapter we will look at another control facility, called 'cut', for preventing backtracking.

5.1 Preventing backtracking

Prolog will automatically backtrack if this is necessary for satisfying a goal. Automatic backtracking is a useful programming concept because it relieves the programmer of the burden of programming backtracking explicitly. On the other hand, uncontrolled backtracking may cause inefficiency in a program. Therefore we sometimes want to control, or to prevent, backtracking. We can do this in Prolog by using the 'cut' facility.

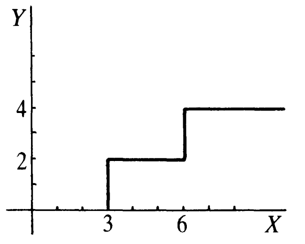

Figure 5.1 A double-step function.

Let us first study the behaviour of a simple example program whose execution involves some unnecessary backtracking. We will identify those points at which the backtracking is useless and leads to inefficiency.

Consider the double-step function shown in Figure 5.1. The relation between X and Y can be specified by three rules:

Rule 1: if X < 3 then Y = 0

Rule 2: if 3 ≦ X and X < 6 then Y = 2

Rule 3: if 6 ≦ X then Y = 4

This can be written in Prolog as a binary relation

```
f( X, Y)
```

as follows:

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

This program, of course, assumes that before f(X, Y) is executed X is already instantiated to a number, as this is required by the comparison operators.

We will make two experiments with this program. Each experiment will reveal some source of inefficiency in the program, and we will remove each source in turn by using the cut mechanism.

5.1.1 Experiment 1

Let us analyze what happens when the following question is posed:

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

When executing the first goal, f(1, Y), Y becomes instantiated to 0. So the second goal becomes

```
2 < 0
```

which fails, and so does the whole goal list. This is straightforward, but before admitting that the goal list is not satisfiable, Prolog tries, through backtracking, two useless alternatives. The detailed trace is shown in Figure 5.2.

The three rules about the f relation are mutually exclusive so that one of them at most will succeed. Therefore we, not Prolog, know that as soon as one rule succeeds there is no point in trying to use the others, as they are bound to fail. In the example of Figure 5.2, rule 1 has become known to succeed at the point indicated by 'CUT'. In order to prevent futile backtracking at this point we have to tell Prolog explicitly *not* to backtrack. We can do this by using the cut mechanism. The 'cut' is written as ! and is inserted between goals as a kind of pseudo-goal. Our program, rewritten with cuts, is:

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

The ! symbol will now prevent backtracking at the points that it appears in the

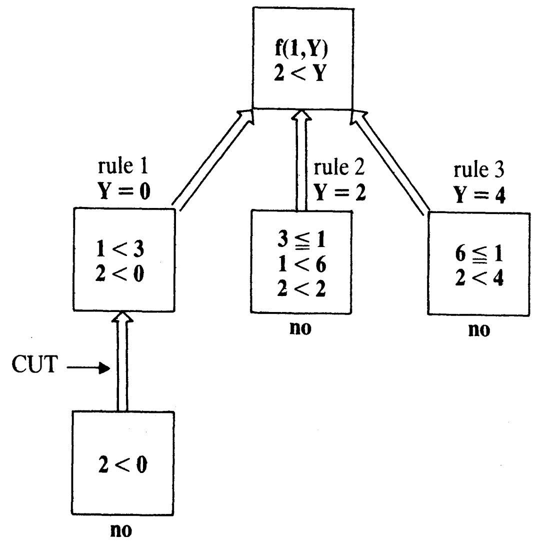

Figure 5.2 At the point marked 'CUT' we already know that the rules 2 and 3 are bound to fail.

program. If we now ask

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

Prolog will produce the same left-hand branch as in Figure 5.2. This branch will fail at the goal 2 < 0. Now Prolog will try to backtrack, but not beyond the point marked ! in the program. The alternative branches that correspond to 'rule 2' and 'rule 3' will not be generated.

The new program, equipped with cuts, is in general more efficient than the original version without cuts. When the execution fails, the new program will in general recognize this sooner than the original program.

To conclude, we have improved the efficiency by adding cuts. If the cuts are now removed in this example, the program will still produce the same result; it will perhaps only spend more time. It can be said that, in our case, by introducing the cut we only changed the procedural meaning of the program; that is, the declarative meaning was not affected. We will see later that using a cut may affect the declarative meaning as well.

5.1.2 Experiment 2

Let us now perform a second experiment with the second version of our program. Suppose we ask:

```
?- f( 7, Y).

Y = 4
```

Let us analyze what has happened. All three rules were tried before the answer was obtained. This produced the following sequence of goals:

Try rule 1: 7 < 3 fails, backtrack and try rule 2 (cut was not reached)

Try rule 2: 3 ≦ 7 succeeds, but then 7 < 6 fails, backtrack and try rule 3 (cut was not reached)

Try rule 3: 6 ≦ 7 succeeds

This trace reveals another source of inefficiency. First it is established that **X < 3** is not true (7 < 3 fails). The next goal is **3 =< X** (3 ≦ 7 succeeds). But we know that once the first test has failed the second test is bound to succeed as it is the negation of the first. Therefore the second test is redundant and the corresponding goal can be omitted. The same is true about the goal **6 =< X** in rule 3. This leads to the following, more economical formulation of the three rules:

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

We can now omit the conditions in the program that are guaranteed to be true whenever they are executed. This leads to the third version of the program:

```
f( X, 0) :- X < 3, !.

f( X, 2) :- X < 6, !.

f( X, 4).
```

This program produces the same results as our original version, but is more efficient than both previous versions. But what happens if we *now* remove the cuts? The program becomes:

```
f( X, 0) :- X < 3.

f( X, 2) :- X < 6.

f( X, 4).
```

This may produce multiple solutions some of which are not correct. For example:

```
?- f( 1, Y).

Y = 0;

Y = 2;

Y = 4;

no
```

It is important to notice that, in contrast to the second version of the program, this time the cuts do not only affect the procedural behaviour, but also change the declarative meaning of the program.

A more precise meaning of the cut mechanism is as follows:

Let us call the 'parent goal' the goal that matched the head of the clause containing the cut. When the cut is encountered as a goal it succeeds immediately, but it commits the system to all choices made between the time the 'parent goal' was invoked and the time the cut was encountered. All the remaining alternatives between the parent goal and the cut are discarded.

To clarify this definition consider a clause of the form:

```
H :- B1, B2, ..., Bm, !, ..., Bn.
```

Let us assume that this clause was invoked by a goal G that matched H. Then G is the parent goal. At the moment that the cut is encountered, the system has already found some solution of the goals **B1**, ..., **Bm**. When the cut is executed, this (current) solution of **B1**, ..., **Bm** becomes frozen and all possible remaining alternatives are discarded. Also, the goal G now becomes committed to this clause: any attempt to match G with the head of some other clause is precluded.

Let us apply these rules to the following example:

```
C :- P, Q, R, !, S, T, U.

C :- V.

A :- B, C, D.

?- A.
```

Here A, B, C, D, P, etc. have the syntax of terms. The cut will affect the execution of the goal C in the following way. Backtracking will be possible within the goal list P, Q, R; however, as soon as the cut is reached, all alternative solutions of the goal list P, Q, R are suppressed. The alternative clause about C,

```
C :- V.
```

will also be discarded. However, backtracking will still be possible within the goal list S, T, U. The 'parent goal' of the clause containing the cut is the goal C in the clause

```
A :- B, C, D.
```

Therefore the cut will only affect the execution of the goal C. On the other hand, it will be 'invisible' from goal A. So automatic backtracking within the goal list B, C, D will remain active regardless of the cut within the clause used for satisfying C.

5.2 Examples using cut

5.2.1 Computing maximum

The procedure for finding the larger of two numbers can be programmed as a relation

```
max( X, Y, Max)
```

where Max = X if X is greater than or equal to Y, and Max is Y if X is less than Y. This corresponds to the following two clauses:

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

max( X, Y, Y)  :-  X < Y.
```

These two rules are mutually exclusive. If the first one succeeds then the second one will fail. If the first one fails then the second must succeed. Therefore a more economical formulation, with 'otherwise', is possible:

If $X \geq Y$ then Max = X,
otherwise Max = Y.

This is written in Prolog using a cut as:

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

max( X, Y, Y).
```

5.2.2 Single-solution membership

We have been using the relation

```
member( X, L)
```

for establishing whether X is in list L. The program was:

```
member( X, [X  |  L] ).

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

This is 'non-deterministic': if X occurs several times then any occurrence can be found. Let us now change **member** into a deterministic procedure which will find only the first occurrence. The change is simple: we only have to prevent backtracking as soon as X is found, which happens when the first clause succeeds. The modified program is:

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

This program will generate just one solution. For example:

```
?- member( X, [a,b,c] ).
X = a;
no
```

5.2.3 Adding an element to a list without duplication

Often we want to add an item X to a list L so that X is added only if X is not yet in L. If X is already in L then L remains the same because we do not want to have redundant duplicates in L. The **add** relation has three arguments

```
add( X, L, L1)
```

where X is the item to be added, L is the list to which X is to be added and L1 is the resulting new list. Our rule for adding can be formulated as:

If X is a member of list L then L1 = L,
otherwise L1 is equal to L with X inserted.

It is easiest to insert X in front of L so that X becomes the head of L1. This is then programmed as follows:

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

The behaviour of this procedure is illustrated by the following example:

```
?- add( a, [b,c], L).
L = [a,b,c]

?- add( X, [b,c], L).
L = [b,c]
X = b
```

```
?- add( a, [b,c,X], L).

L = [b,c,a]
X = a
```

This example is instructive because we cannot easily program the 'non-duplicate adding' without the use of cut or another construct derived from the cut. If we omit the cut in the foregoing program then the **add** relation will also add duplicate items. For example:

```
?- add( a, [a,b,c], L).

L = [a,b,c];

L = [a,a,b,c]
```

So the cut is necessary here to specify the right relation, and not only to improve efficiency. The next example also illustrates this point.

5.2.4 Classification into categories

Assume we have a database of results of tennis games played by members of a club. The pairings were not arranged in any systematic way, so each player just played some other players. The results are in the program represented as facts like:

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

We want to define a relation

```
class( Player, Category)
```

that ranks the players into categories. We have just three categories:

winner: every player who won all his or her games is a winner

fighter: any player that won some games and lost some

sportsman: any player who lost all his or her games

For example, if all the results available are just those above then Ann and Pat are winners, Tom is a fighter and Jim is a sportsman.

It is easy to specify the rule for a fighter:

```
X is a fighter if
  there is some Y such that X beat Y and
  there is some Z such that Z beat X.
```

Now a rule for a winner:

```
X is a winner if
  X beat some Y and
  X was not beaten by anybody.
```

This formulation contains 'not' which cannot be directly expressed with our present Prolog facilities. So the formulation of **winner** appears trickier. The same problem occurs with **sportsman**. The problem can be circumvented by combining the definition of **winner** with that of **fighter**, and using the 'otherwise' connective. Such a formulation is:

```
If X beat somebody and X was beaten by somebody
  then X is a fighter,
  otherwise if X beat somebody
    then X is a winner,
    otherwise if X got beaten by somebody
      then X is a sportsman.
```

This formulation can be readily translated into Prolog. The mutual exclusion of the three alternative categories is indicated by the cuts:

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

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

class( X, sportsman) :-
  beat( _, X).
```

Exercises

5.1 Let a program be:

```
p( 1).
p( 2) :- !.
p( 3).
```

Write all Prolog's answers to the following questions:

(a) ?- p(X).

(b) ?- p(X), p(Y).

(c) ?- p(X), !, p(Y).

5.2 The following relation classifies numbers into three classes: positive, zero and negative:

```
class( Number, positive) :-  Number > 0.
class( 0, zero).
class( Number, negative) :-  Number < 0.
```

Define this procedure in a more efficient way using cuts.

5.3 Define the procedure

```
split( Numbers, Positives, Negatives)
```

which splits a list of numbers into two lists: positive ones (including zero) and negative ones. For example,

```
split( [3,-1,0,5,-2], [3,0,5], [-1,-2] )
```

Propose two versions: one with a cut and one without.

5.3 Negation as failure

'Mary likes all animals but snakes'. How can we say this in Prolog? It is easy to express one part of this statement: Mary likes any X if X is an animal. This is in Prolog:

```
likes( mary, X) :-  animal( X).
```

But we have to exclude snakes. This can be done by using a different formulation:

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

That something is not true can be said in Prolog by using a special goal, **fail**, which always fails, thus forcing the parent goal to fail. The above formulation is translated into Prolog, using **fail**, as follows:

```
likes( mary, X) :-
  snake( X), !, fail.

likes( mary, X) :-
  animal( X).
```

The first rule here will take care of snakes: if X is a snake then the cut will prevent backtracking (thus excluding the second rule) and **fail** will cause the failure. These two clauses can be written more compactly as one clause:

```
likes( mary, X) :-
  snake( X), !, fail;
  animal( X).
```

We can use the same idea to define the relation

different(X, Y)

which is true if X and Y are different. We have to be more precise, however, because '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 expressions X and Y are not equal.

Let us choose here that X and Y are different if they do not match. The key to saying this in Prolog is:

If X and Y match then **different(X, Y)** fails,
otherwise **different(X, Y)** succeeds.

We again use the cut and **fail** combination:

```
different( X, X) :- !, fail.

different( X, Y).
```

This can also be written as one clause:

```
different( X, Y) :-
  X = Y, !, fail;
  true.
```

true is a goal that always succeeds.

These examples indicate that it would be useful to have a unary predicate 'not' such that

not(Goal)

is true if **Goal** is not true. We will now define the **not** relation as follows:

If **Goal** succeeds then **not(Goal)** fails,
otherwise **not(Goal)** succeeds.

This definition can be written in Prolog as:

```
not( P) :-
  P, !, fail;
  true.
```

Henceforth, we will assume that **not** is a built-in Prolog procedure that behaves as defined here. We will also assume that **not** is defined as a prefix operator, so that we can also write the goal

```
not( snake(X) )
```

as:

```
not snake( X)
```

Many Prolog implementations do in fact support this notation. If not, then we can always define **not** ourselves.

It should be noted that **not** defined as failure, as here, does not exactly correspond to negation in mathematical logic. This difference can cause unexpected behaviour if **not** is used without care. This will be discussed later in the chapter.

Nevertheless, **not** is a useful facility and can often be used advantageously in place of cut. Our two examples can be rewritten with **not** as:

```
likes( mary, X) :-
  animal( X),
  not snake( X).

different( X, Y) :-
  not ( X = Y).
```

This certainly looks better than our original formulations. It is more natural and is easier to read.

Our tennis classification program of the previous section can also be rewritten, using **not**, in a way that is closer to the initial definition of the three categories:

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

class( X, winner) :-
  beat( X, _),
  not beat( _, X).
```

```
class( X, sportsman) :-
  beat( _, X),
  not beat( X, _).
```

As another example of the use of **not** let us reconsider program 1 for the eight queens problem of the previous chapter (Figure 4.7). We specified the **no_attack** relation between a queen and other queens. This relation can be formulated also as the negation of the attack relation. Figure 5.3 shows a program modified accordingly.

Exercises

5.4 Given two lists, **Candidates** and **RuledOut**, write a sequence of goals (using **member** and **not**) that will through backtracking find all the items in **Candidates** that are not in **RuledOut**.

5.5 Define the set subtraction relation

```
difference( Set1, Set2, SetDifference)
```

where all the three sets are represented as lists. For example:

```
difference( [a,b,c,d], [b,d,e,f], [a,c] )
```

```
solution( [] ).

solution( [X/Y | Others] ) :-
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  not attacks( X/Y, Others).

attacks( X/Y, Others) :-
  member( X1/Y1, Others),
  ( Y1 = Y;
    Y1 is Y + X1 - X;
    Y1 is Y - X1 + X ).

member( A, [A | L] ).

member( A, [B | L] ) :-
  member( A, L).

% Solution template

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

Figure 5.3 Another eight queens program.

5.6 Define the predicate

unifiable(List1, Term, List2)

where **List2** is the list of all the members of **List1** that match **Term**, but are not instantiated by this matching. For example:

```
?- unifiable( [X, b, t(Y)], t(a), List] ).
List = [X, t(Y)]
```

Note that X and Y have to remain uninstantiated although the matching with t(a) does cause their instantiation. Hint: Use **not (Term1 = Term2)**. If **Term1 = Term2** succeeds then **not (Term1 = Term2)** fails and the resulting instantiation is undone!

5.4 Problems with cut and negation

Using the cut facility we get something, but not for nothing. The advantages and disadvantages of using cut were illustrated by examples in the previous sections. Let us summarize, first the advantages:

(1) With cut we can often improve the efficiency of the program. The idea is to explicitly tell Prolog: do not try other alternatives because they are bound to fail.

(2) Using cut we can specify mutually exclusive rules; so we can express rules of the form:

if condition P *then* conclusion Q,
otherwise conclusion R

In this way, cut enhances the expressive power of the language.

The reservations against the use of cut stem from the fact that we can lose the valuable correspondence between the declarative and procedural meaning of programs. If there is no cut in the program we can change the order of clauses and goals, and this will only affect the efficiency of the program, not the declarative meaning. On the other hand, in programs with cuts, a change in the order of clauses may affect the declarative meaning. This means that we can get different results. The following example illustrates:

```
p :- a, b.
p :- c.
```

The declarative meaning of this program is: p is true if and only if a and b are both true or c is true. This can be written as a logic formula:

$$p \quad <===> \quad (a \,\&\, b) \vee c$$

We can change the order of the two clauses and the declarative meaning remains the same. Let us now insert a cut:

```
p :- a, !, b.
p :- c.
```

The declarative meaning is now:

$$p \iff (a \& b) \lor (\sim a \& c)$$

If we swap the clauses,

```
p :- c.
p :- a, !, b.
```

then the meaning becomes:

$$p \iff c \lor (a \& b)$$

The important point is that when we use the cut facility we have to pay more attention to the procedural aspects. Unfortunately, this additional difficulty increases the probability of a programming error.

In our examples in the previous sections we have seen that sometimes the removal of a cut from the program can change the declarative meaning of the program. But there were also cases in which the cut had no effect on the declarative meaning. The use of cuts of the latter type is less delicate, and therefore cuts of this kind are sometimes called 'green cuts'. From the point of view of readability of programs, green cuts are 'innocent' and their use is quite acceptable. When reading a program, green cuts can simply be ignored.

On the contrary, cuts that do affect the declarative meaning are called 'red cuts'. Red cuts are the ones that make programs hard to understand, and they should be used with special care.

Cut is often used in combination with a special goal, **fail**. In particular, we defined the negation of a goal (**not**) as the failure of the goal. The negation, so defined, is just a special (more restricted) way of using cut. For reasons of clarity we will prefer to use **not** instead of the *cut–fail* combination (whenever possible), because the negation is a higher level concept and is intuitively clearer than the *cut–fail* combination.

It should be noted that **not** may also cause problems, and so should also be used with care. The problem is that **not**, as defined here, does not correspond exactly to negation in mathematics. If we ask Prolog

```
?- not human( mary).
```

Prolog will probably answer 'yes'. But this should not be understood as Prolog saying 'Mary is not human'. What Prolog really means to say is: 'There is not enough information in the program to prove that Mary is human'. This arises because when processing a **not** goal, Prolog does not try to prove this goal directly. Instead, it tries to prove the opposite, and if the opposite cannot be proved then Prolog assumes that the **not** goal succeeds. Such reasoning is based on the so-called *closed world assumption*. According to this assumption *the world is closed* in the sense that everything that exists is in the program or can be derived from the program. Accordingly then, if something is not in the program (or cannot be derived from it) then it is not true and consequently its negation is true. This deserves special care because we do not normally assume that 'the world is closed': with not explicitly entering the clause

```
human( mary).
```

into our program, we do not normally mean to imply that Mary is not human.

We will, by example, further study the special care that **not** requires:

```
r( a).

q( b).

p( X) :- not r( X).
```

If we now ask

```
?- q( X), p( X).
```

then Prolog will answer

```
X = b
```

If we ask apparently the same question

```
?- p( X), q( X).
```

then Prolog will answer:

```
no
```

The reader is invited to trace the program to understand why we get different answers. The key difference between both questions is that the variable X is, in the first case, already instantiated when **p(X)** is executed, whereas at that point X is not yet instantiated in the second case.

We have discussed problems with cut, which also indirectly occur in **not**, in detail. The intention has been to warn users about the necessary care, not to definitely discourage the use of cut. Cut is useful and often necessary. And after all, the kind of complications that are incurred by cut in Prolog commonly occur when programming in other languages as well.

Summary

- The cut facility prevents backtracking. It is used both to improve the efficiency of programs and to enhance the expressive power of the language.
- Efficiency is improved by explicitly telling Prolog (with cut) not to explore alternatives that we know are bound to fail.
- Cut makes it possible to formulate mutually exclusive conclusions through rules of the form:

  if **Condition** *then* **Conclusion1** *otherwise* **Conclusion2**
- Cut makes it possible to introduce *negation as failure*: **not Goal** is defined through the failure of **Goal**.
- Two special goals are sometimes useful: **true** always succeeds, **fail** always fails.
- There are also some reservations against cut: inserting a cut may destroy the correspondence between the declarative and procedural meaning of a program. Therefore, it is part of good programming style to use cut with care and not to use it without reason.
- **not** defined through failure does not exactly correspond to negation in mathematical logic. Therefore, the use of **not** also requires special care.

Reference

The distinction between 'green cuts' and 'red cuts' was proposed by van Emden (1982).

van Emden, M. (1982) Red and green cuts. *Logic Programming Newsletter*: **2**.