Deductive Databases and Logic Programming

(Winter 2003/2004)

Chapter 5: Practical Prolog Programming

- The Cut and Related Constructs
- Prolog vs. Pascal
- Definite Clause Grammars

Objectives

After completing this chapter, you should be able to:

- explain the effect of the cut.
- write Prolog programs for practical applications.
- use context-free grammars in Prolog.

Overview

1. The Cut and Related Constructs
2. Prolog vs. Pascal
3. Definite Clause Grammars

The Cut: Effect (1)

- The cut, written "!" in Prolog, removes alternatives that otherwise would have been tried during backtracking. E.g. consider this rule:

  $p(t_1, \ldots, t_k)$:- A_1, ..., A_m, !, B_1, ..., B_n.
- Until the cut is executed, processing is as usual.
- When the cut is reached, all previous alternatives for this call to the predicate p are removed:
  - ◇ No other rule about p will be tried.
  - ◇ No other solutions to the literals A, ..., A_m will be considered.

The Cut: Effect (2)

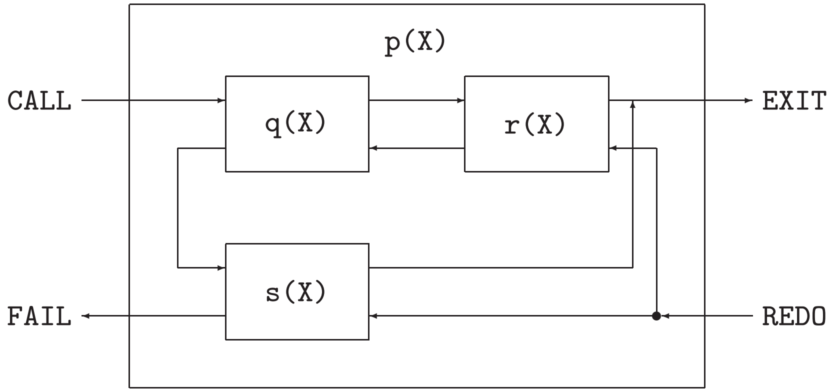
- Example:

```
p(X) :- q(X), !, r(X).
p(X) :- s(X).
q(a).
q(b).
r(X).
s(c).
```

- With the cut, the query `p(X)` returns only `X=a`.
- Without the cut, the solutions are `X=a`, `X=b`, `X=c`.
- Exercise: Can the second rule about `p` ever be used?

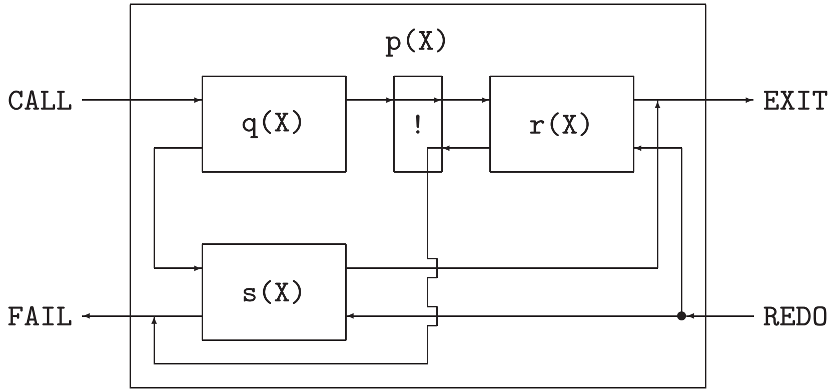
The Cut: Effect (3)

Four-Port Model without Cut:



The Cut: Effect (4)

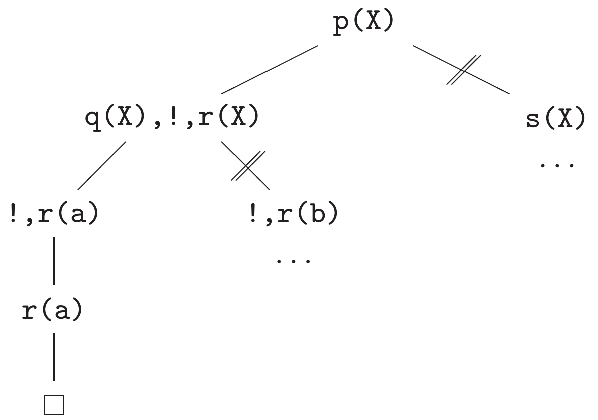
Four-Port Model with Cut:



The Cut: Effect (5)

- A call to the cut immediately succeeds (like `true`).
- Any try to redo the cut not only fails, but immediately fails the entire predicate call.
- In the SLD-tree, the cut "cuts away" all still open branches between
  - ◇ the node where the cut was introduced (i.e. the child of which contains the cut), and
  - ◇ the node where the cut is the selected literal.

The Cut: Effect (6)



The Cut: Effect (7)

- Normal backtracking before and after the cut (only not through the cut):

```
p(X,Y) :- q1(X), r1(X,Y).
p(X,Y) :- q1(X), q2(X), !, r1(X,Y), r2(Y).
p(X,Y) :- s(X,Y).
q1(a).
q1(b).     q2(b).
r1(b,c).
r1(b,d).   r2(d).
s(e,f).
```

- The query `p(X,Y)` has solutions `X=b,Y=c` and `X=b,Y=d`.

Cut: Improving Runtime (1)

- One application of the cut is to improve the runtime of a program by eliminating parts of the proof tree that cannot yield solutions or at least cannot yield any new solutions.
- Consider the predicate `abs` that computes the absolute value of a number:

```
abs(X,X) :- X >= 0.
abs(X,Y) :- X =< 0, Y is -X.
```

- When the first rule is successful, it is clear that the second rule does not have to be tried.

Cut: Improving Runtime (2)

- Consider now the goal `p(X), abs(X,Y), Y > 5` with the facts `p(3)`, `p(0)`, `p(-7)`.
- First `p(X)` succeeds with `X` bound to `3`, then `abs(3,Y)` succeeds for `Y=3`, but then `3 > 5` fails.
- Now backtracking would normally first try to find an alternative solution for `abs(3,Y)`, since there is another rule about `abs` that has not yet been tried.
- This is obviously useless, and the runtime can be improved by immediately backtracking to `p(X)`.

Cut: Improving Runtime (3)

- With the cut, one can tell the Prolog system that when the first rule succeeds, the second rule cannot give anything new:

```
abs(X,X) :- X >= 0, !.
abs(X,Y) :- X =< 0, Y is -X.
```

- Of course, one could have (should have) written the condition in the second rule `X < 0`.
- Then some (but not all) Prolog systems are able to discover themselves that the rules are mutually exclusive.

Cut: Improving Space (1)

- Making clear that a predicate has no other solution improves also the space (memory) efficiency.
- The Prolog system must keep a record ("choice-point") for each predicate call that is not yet complete (for backtracking into the predicate call later).
- Even worse, certain data structures within the Prolog system must be "frozen" when it is necessary to support later backtracking to this state.
- Then e.g. variable bindings must be logged (on the "trail") so that they can later be undone.

Cut: Improving Space (2)

- In imperative languages, when a procedure call returns, its stack frame (containing local variables and other information) can be reused.
- In Prolog, this is not always the case, because it might be necessary to reactivate the procedure call and search for another solution.
- E.g. consider the following program:

```
p(X) :- q(X), r(X).
q(X) :- s(X), t(X).
s(a). s(b). t(a). t(b). r(b).
```


Cut: Improving Space (3)

- The call `q(X)` first exits with `X=a`, but then `r(a)` fails, thus the call `q(X)` is entered again, which in turn reactives `s(X)`.

  Upon backtracking, also the binding of `X` must be undone.

- In the above example, not much can be improved, because there really are alternative solutions.
- However, when a predicate call has only one solution, it should be executed like a procedure call in an imperative language.

Cut: Improving Space (4)

- Predicate calls that can have at most one solution are called deterministic.

  Sometimes one calls the predicate itself deterministic, but then one usually has a specific binding pattern in mind. E.g. `append` is deterministic for the binding pattern `bbf`, but it is not deterministic for `ffb`.

- For efficient execution, it is important that the Prolog system understands that a predicate call is deterministic. Here a cut can help.

  Actually, the cut in the definition of `abs` makes the predicate deterministic. In general, it might be important that `abs(0,X)` succeeds "two times", Prolog is not allowed to automatically remove one solution. Deductive databases are set-oriented, there more powerful optimizers are possible.

Cut: Improving Space (5)

- Consider `abs` applied to a list:

```
abs_list([], []).
abs_list([X|R], [Y|S]) :- abs(X, Y),
                          abs_list(R, S).
```

- When the Prolog system thinks that `abs` is nondeterministic, it will keep the stackframe for each call to `abs` (and for the calls to `abs_list`).
- When a predicate calls a nondeterministic predicate, it automatically becomes nondeterministic, too.

  Only for the last body literal of the last rule about a predicate, the stack frame of the predicate is reused (under certain conditions), and thus does not remain, even when this body literal is non-deterministic.

Cut: Improving Space (6)

- In the above example, making `abs` deterministic (by means of a cut) is a big improvement.
- Then most Prolog systems will automatically deduce that also `abs_list` is deterministic.

  For the only possible binding patterns `bf` and `bb`.

- Usually, the outermost functor of the first argument is considered: Since it is "`[]`" for the first rule, and "`.`" for the second, always only one of the two rules is applicable (if the first argument is bound).

Cut: Improving Space (7)

- It is also possible to remove unnecessary stack frames at a later point.
- E.g. suppose that `abs` (and thus `abs_list`) remain nondeterministic, and consider the goal:

```
abs_list([-3,7,-4], X), !, p(X).
```

- The call to `abs_list` will leave many stack frames behind, but these are deleted by the cut.

  It is probably better style to avoid the nondeterminism at the place where it occurs. However, one should not use too many cuts, and it might be easier to clean up the stack only at a few places.

Cut: If-Then-Else (1)

- The cut is also used to encode an "if then else".
- Consider the following predicate:

```
p(X, Y) :- q1(X), !, r1(X, Y).
p(X, Y) :- q2(X), !, r2(X, Y).
p(X, Y) :- r3(X, Y).
```

- This is equivalent to (assuming that q1 and q2 are deterministic):

```
p(X, Y) :- q1(X), r1(X, Y).
p(X, Y) :- \+ q1(X), q2(X), r2(X, Y).
p(X, Y) :- \+ q1(X), \+ q2(X), r3(X, Y).
```


Cut: If-Then-Else (2)

- The formulation with the cut is a bit shorter.

  The difference becomes the bigger, the more cases there are.

- Furthermore, the runtime is shorter: In the version without the cut, q1(X) is computed up to three times.
- But removing the cut in first version would completely change the semantics of the program.

  The cut is no longer only an "optimizer hint".

Cut: If-Then-Else (3)

- The logical semantics of programs with negation as failure ("\+") has be extensively studied and there are good proposals.
- I do not know of successful tries to give the cut a clear logical (declarative) semantics.

  The cut can basically be understood only operationally. One problem is that the cut is used for many different purposes, and it might be difficult to automatically discover for which one.

- Pure logic programmers try to avoid the cut, at least when it affects the logic of the program.

Cut: If-Then-Else (4)

- Prolog has an "if-then" operator -> that can be used to have the advantages of the cut, while making the logical intention clear.
- E.g. one could write the above procedure as

```
p(X, Y) :- q1(X) -> r1(X,Y);
           q2(X) -> r2(X,Y);
           r3(X,Y).
```

- A -> B has basically the same effect as A, !, B.

  However, if there should be further rules about p, this cut does not remove the possibility to try these rules. It does remove alternative solutions for A, and it does remove the possibility to try the disjunctive alternatives within the rule.

Cut: Negation

- Conversely, one can implement negation as failure with the cut (`not` is only another name for `\+`):

```
not(A) :- call(A), !, fail.
not(_).
```

- The first rule ensures that if `A` succeeds, `not(A)` fails.
- The second rule makes `not(A)` true in all other cases (i.e. when `A` fails).

  Of course, if `A` should run into an infinite loop, also `not(A)` does not terminate.

Cut: One Solution (1)

- Suppose that email addresses of professors are stored as facts, and that the same person can have several email addresses:

```
prof_email(brass, 'sbrass@sis.pitt.edu').
prof_email(brass, 'brass@acm.org').
prof_email(spring, 'mspring@sis.pitt.edu').
...
```

- The cut can be used to select a single address of a given professor:

```
prof_email(brass, E), !, send_email(E).
```


Cut: One Solution (2)

- Prolog has a built-in predicate `once` that can be used instead of the cut:

```
once(prof_email(brass, E)), send_email(E).
```

- `once` is defined as:

  `once(`A`) :- call(`A`), !.`

- In the example, the following is equivalent:

```
prof_email(brass, E) -> send_email(E).
```

  However, the solution with `once` makes the intention clearer.

Cut: Dangers (1)

- The cut can make programs wrong if predicates are called with unexpected binding patterns.
- E.g. the predicate for the absolute value can also be written as follows (using the cut as in the if-then-else pattern):

```
abs(X,X) :- X >=0, !.
abs(X,Y) :- Y is -X.
```

- Since the second rule is executed only when the first rule is not applicable, it might seem that the test `X =< 0` used earlier is superfluous.

Cut: Dangers (2)

- This is indeed true for the binding pattern `bf`, but consider now the call `abs(3,-3)`!
- In general, the rule is that the cut must be exactly at the point where it is clear that this is the right rule: Not too early and not too late.
- Here the unification must happen after the cut:

```
abs(X,Y) :- X >= 0, !, X = Y.
abs(X,Y) :- Y is -X.
```

- This would work also with binding pattern `bb`.

Cut: Dangers (3)

- Consider this predicate:

```
person(X, male)   :- man(X), !.
person(X, female) :- woman(X).
```

- Since `man` and `woman` are disjoint, the cut was only added to improve the efficiency.
- It works if `person` is called with binding pattern `bf` or `bb`. However, consider what happens if `person` is called with binding pattern `ff`!

  It is interesting that here the more general binding pattern poses a problem, whereas in the `abs` example, the more specific binding pattern is not handled.

Types of Cuts

- Cuts in Prolog programs are usually classified into
  - ◇ Green Cuts: Do not modify the logical meaning of the program, only improve the runtime/space efficiency.

    Some authors also distinguish blue cuts: In this case, a good Prolog system should be able to determine itself that there are no further solutions. Blue cuts are intended only for very simple (dump) Prolog systems. "Grue Cuts": Green or blue cuts.
  - ◇ Red cuts: Modify the declarative meaning of the program.

    Good Prolog programmers try to use red cuts only very seldom.

Cut: Summary, Outlook

- The cut is necessary for efficient Prolog programming, but it destroys the declarative meaning of the programs and can have unexpected consequences.
- The better Prolog implementations get, the less important will be the cut.
- Newer logic programming languages usually try to replace the cut by other constructs that have a more declarative meaning.
- If possible, use `->`, `\+`, `once` instead.
- Use the cut only as last resort.