Logic Programming: Non-logical features

Alan Smaill

Oct 1 2015

Note

- Several predicates seen so far or today are built-in in sicstus, maybe with different names.
  - `append/3`
  - `member/2`
  - `length/2`
- It is good to know how to define them from scratch, if necessary.
- LPN "predicate index" lists all the built-ins you are expected to know, and more . . .

Nonlogical features

So far we have worked mostly in pure Prolog.
This provides:

- solid logical basis
- elegant solution to symbolic problems

But various practical things become inconvenient:
arithmetic and I/O.

And standard proof search is not always efficient.

- Can we control proof search better?

Today's agenda

- Nonlogical features
  - Expression evaluation
  - I/O
  - "Cut" — pruning the search space
  - Negation as failure

Expression evaluation

- Prolog has built-in syntax for arithmetic expressions;
- But it is uninterpreted – simply syntax.

```
?- 2 + 2 = 4.

no.

?- X = 2+2.

X=2+2

?- display(2+2).

+(2,2)
```

Expression evaluation

We can define unary arithmetic operations ourselves:

```
add(M,N,P)
```

and define evaluation ourselves:

```
eval(+(X, Y), V) :-
   eval(X, N), eval(Y, M), add(M, N, V).
```

but this is painfully slow —
and floating point would be worse.

evaluation predicate "*is*"

```
?- X is 2+2.

X=4.

?- X is 6*7.

X=42.

?- X is 2+Y.
! Instantiation error in argument 2 of is/2
! goal:  _107 is 2+_111
```

Machine Arithmetic with "is"

- Addition (+), subtraction(-)

```
X is 2+(3-1).

X=4
```

- multiplication (*), division(/), mod:

```
?- X is 42 mod 5, Y is 42 / 5.

X = 2,
Y = 8.4
```

WARNING

- Unlike "=", "is" is **not** symmetric.
- needs the mode (?,+)
  so requires RHS to be **ground** (no variables):

```
?- 2+(3-1) is X.

! Instantiation error i...
```

- Further, the RHS must be an arithmetic expression:

```
?- X is foo(z).

! Domain error ...
```

Lists and arithmetic

- length of a list:
  possible definition:

```
len([], 0).
len([_|L], N) :- len(L, M), N is M+1.
```

- Only works in mode `(+,?)`
- The built-in (in sicstus) `length/2` works in both directions.

Arithmetic comparisons

There are several binary relations built-in as goals, written infix:

- less than (<), greater than (>)
- less or equal (=<), greater or equal (>=)
- arithmetic equality (`=:=`), inequality (`=/=`)

All of these have mode `(+,+)`:
both arguments must be **ground**.

Example

Maximum predicate (3rd arg is max of other two)

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

Works in mode `(+X, +Y, ?M)`

Basic Input/Output

- `read(?X)` reads in a term, by default from standard input;
  – the term must be followed by a "."
- `write(+X)` prints out its argument as a term;
  – if `X` is not ground, variable names are not preserved.
- `nl/0` prints a newline.

Expression calculator, taking input from terminal:
note non-terminating loop!

```
calc :- read(X),
        Y is X,
        write(X = Y), nl,
        calc.
```

How do backtracking and I/O interact?

Short answer: backtracking is possible, but cannot undo I/O.

```
?- write(foo), fail; write(bar).

foobar
```

As is normal, any binding is undone on backtracking:

```
?- read(X), fail; X=1.

|:  foo.

X=1.
```

Cut

- Sometimes, we have reason to believe we have reached the right / only possible answer
  - so no back-tracking is needed
- in Pure Prolog, we cannot take advantage of this
- Introduce a special "cut" predicate to allow this to be expressed.
- Cut just written by exclamation mark: !

Example

The "member of a list" predicate:

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

If this is used in mode `(+,+)`, and `X` is found in `L`, there is no point in backtracking and looking for other solutions.

Example

So, insert a cut in the first clause:

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

When a goal that matches `member(X,Y)` is called, if the first clause succeeds, the second will not be used on backtracking.

How it works

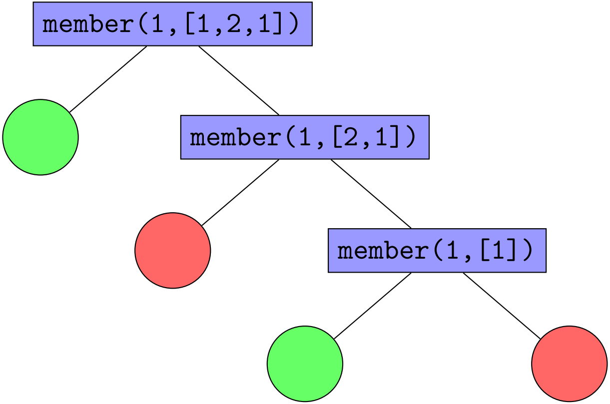
Recall that there is a choice point any time there are alternative clauses for using a clause for a particular (atomic) goal.

Suppose there is a cut in the body of some clause of predicate `pred/2`; and an attempt to solve sub-goal `pred(T1,T2)` has reached the cut. Then:

- as a goal, the cut succeeds
- and, while solving `pred(T1,T2)`, it cuts out ("prunes") any remaining choice points:
  - earlier in the body of that clause, and
  - cuts out all later clauses of `pred/2`.

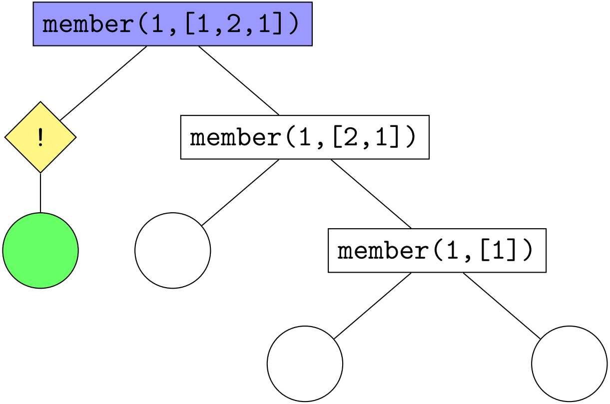
How it works

without cut



How it works

with cut



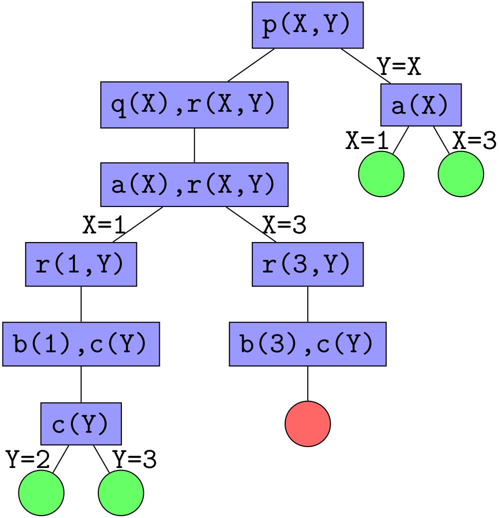
```
p(X,Y):-
  q(X),r(X,Y).
p(X,X):- a(X).

q(X):- a(X).

r(X,Y):-
  b(X),c(Y).

a(1).  a(3).
b(1).  b(2).
c(2).  c(3).

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



examples (2)

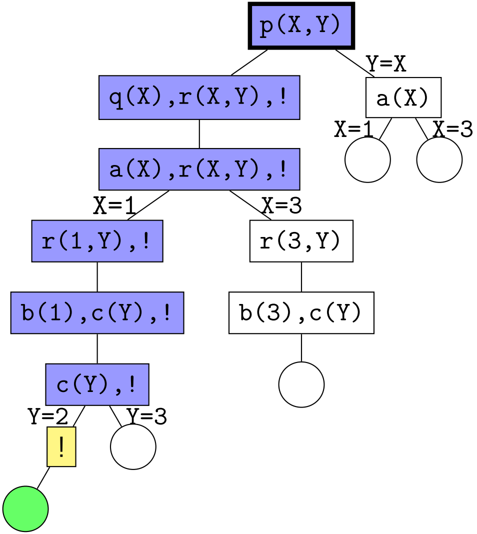
```
p(X,Y):-
  q(X),r(X,Y), !.
p(X,X):- a(X).

q(X):- a(X).

r(X,Y):-
  b(X),c(Y).

a(1).  a(3).
b(1).  b(2).
c(2).  c(3).

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



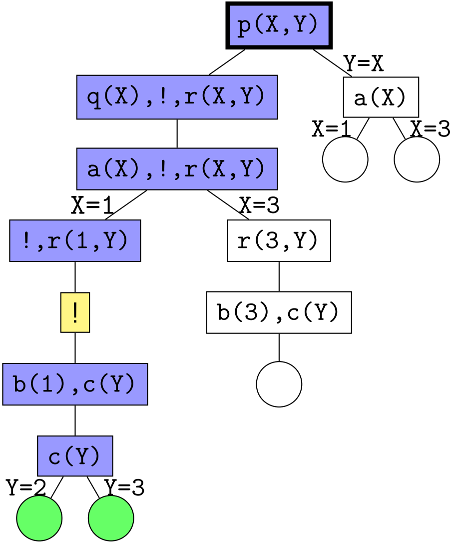
```
p(X,Y):-
   q(X),!,r(X,Y).
p(X,X):- a(X).

q(X):- a(X).

r(X,Y):-
   b(X),c(Y).

a(1).   a(3).
b(1).   b(2).
c(2).   c(3).

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



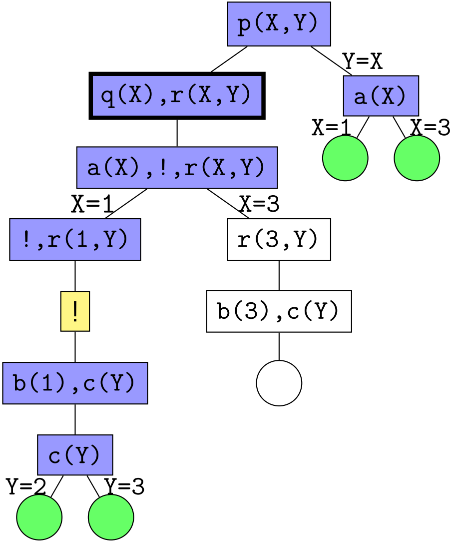
```
p(X,Y):-
   q(X),r(X,Y).
p(X,X):- a(X).

q(X):- a(X), !.

r(X,Y):-
   b(X),c(Y).

a(1).  a(3).
b(1).  b(2).
c(2).  c(3).

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



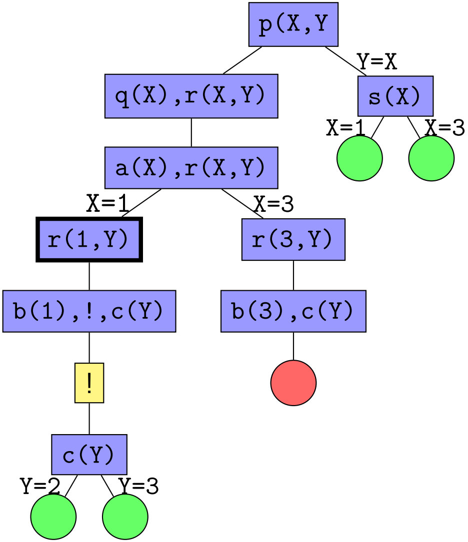
```
p(X,Y):-
  q(X),r(X,Y).
p(X,X):- a(X).

q(X):- a(X).

r(X,Y):-
  b(X),!,c(Y).

a(1).  a(3).
b(1).  b(2).
c(2).  c(3).

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



Max

Our earlier implementation:

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

- It is pointless to backtrack –
  - if the first clause succeeds, then the second must fail.

Max using cut

So stop backtracking:

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

- It is pointless to backtrack –
  - if the first clause succeeds, then the second must fail.

But what about . . .

Do we need the test in the second clause at all ?
Let's try dropping it:

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

This is (slightly) more efficient, but

- it damages transparency — not the right logical characterisation.
- `max(1,2,1)` and `max(1,2,2)` both succeed! (Why?)
- clause order matters!

Safe use of cut

- Cut can make programs more efficient
  - by avoiding pointless backtracking
- But as we have just seen with `max/3`, cuts can change the meaning of the program (not just efficiency).
- "Green" cuts are those that preserve meaning of the program;
- "Red" cuts don't.

Forthcoming attractions

- More about cut and negation
- Further reading: LPN chs 5 & 10.