Logic Programming: Negation as failure, sets, terms

Alan Smaill

Oct 12 2015

- Non-logical features ctd
- Negation as Failure
- Collecting solutions (findall, setof, bagof)
- Assert and retract
- Processing terms

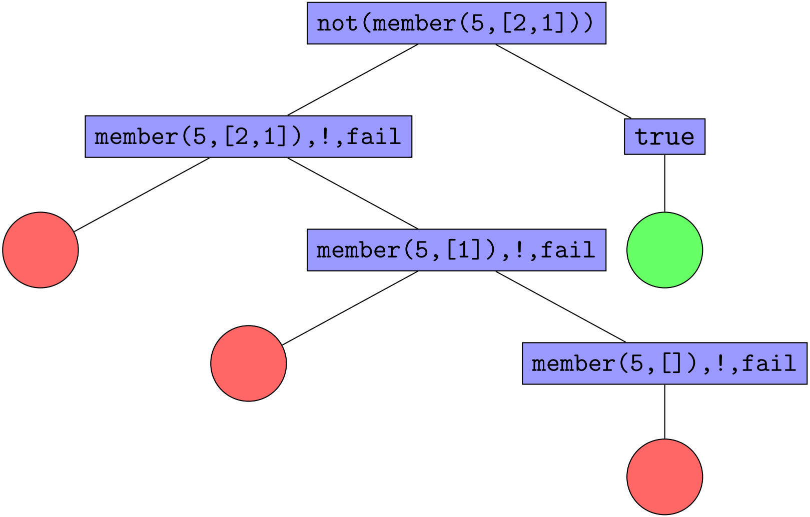
Negation as Failure

- We can use cut to define negation as failure
  - Recall first tutorial:

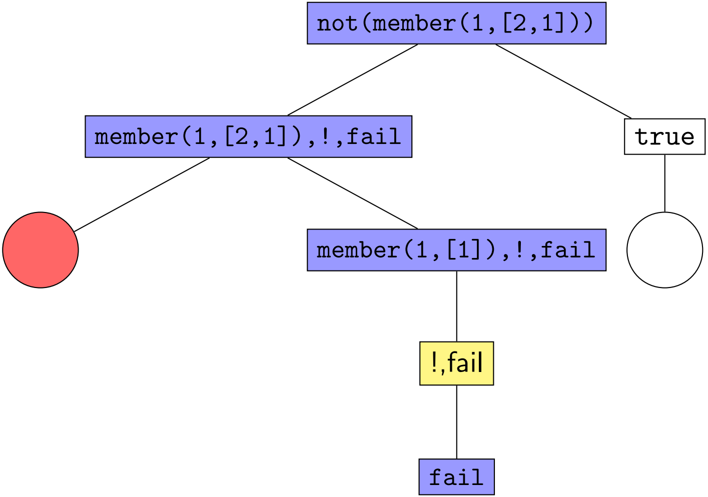
```
not(G) :- G, !, fail; true.
```

- This tries to solve G:
  - if successful, fail;
  - otherwise succeed.

How it works



not(member(5,[2,1]))
member(5,[2,1]),!,fail
true
member(5,[1]),!,fail
member(5,[]),!,fail


not(member(1,[2,1]))
member(1,[2,1]),!,fail
true
member(1,[1]),!,fail
!,fail
fail

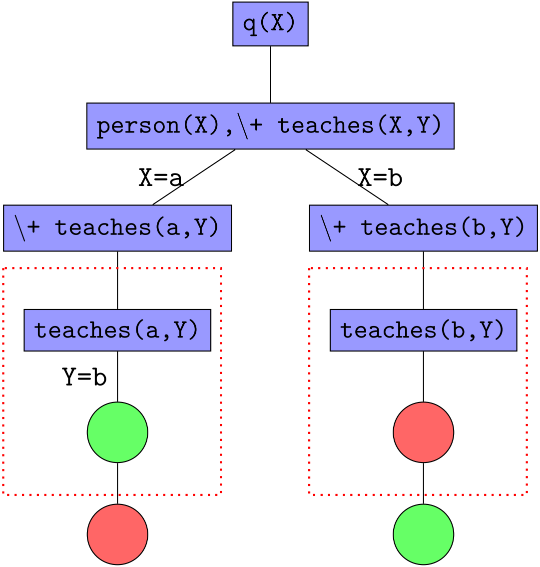
Negation as Failure

- Built-in syntax: `\+ G`
- Example: people who are not teachers:

```
q(X) :- person(X),
        \+ teach(X,Y).
```

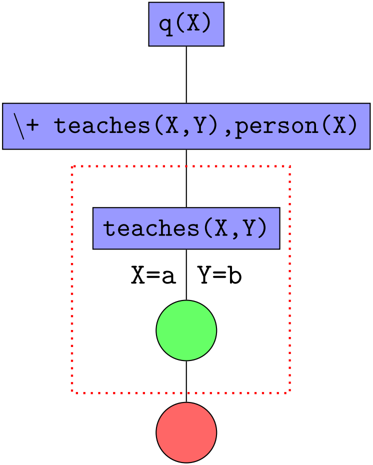
```
person(a).
person(b).
teach(a,b).

q(X) :-
  person(X),
\+ teach(X,Y).
```



```
person(a).
person(b).
teach(a,b).

q(X) :-
  \+ teach(X,Y),
  person(X),
```



The second order above shows **non-logical** behaviour; negation as failure is not logical negation.

Goal order matters

- This order fails:

  ```
  ?- \+ X = Y, X = a, Y = b .
  ```

- This order succeeds:

  ```
  ?- X = a, Y = b, \+ X = Y.
  ```

  Since comma corresponds to conjunction, the declarative reading would say that the two queries are logically equivalent; so if one succeeds, the other cannot fail.

Safe use of negation as failure

- We can read `\+ G` as the logical "not G" only if `G` is ground when we start solving it.
- Any free variables are treated as "existentially quantified":
  - `?- \+ 1 = 2.` is treated as $\neg(1 = 2)$
  - `?- \+ X = Y.` is treated as $\neg(\exists X\ \exists Y\ X = Y)$.
- HEURISTIC: delay negation after other goals to allow negated goals to become ground.

Sometimes we want to find **all** solutions for a given query,
eg collected as an explicit list –
which had better be finite.

- Want something like `alist(bart,X)` to find `X` which lists all the ancestors of `bart`.
- Can't do this in pure Prolog –
  cut is not helpful.
- Technically possible (but painful) using assert/retract.

Collecting solutions declaratively

There are built-in procedures to do this:

`findall/3` builds a list of solutions:

```
?- findall(Y, ancestor(Y, bart), L).
L = [homer,marge,abe,jacqueline]

?- findall((X,Y), ancestor(X,Y), L).
L = [(abe,homer), (homer,bart), (homer,lisa) | ...]
```

findall

Usage:

```
findall(?X, ?Goal, ?List)
```

- On success, `List` is list of all substitutions for `X` for which `Goal` succeeds.
- The Goal can have free variables
  - but `X` is treated as "bound" in `Goal`
- `X` can also be a pattern, as in second example above.

bagof/3 also computes a list of solutions:

```
?- bagof(Y, ancestor(Y, bart), L).
L = [homer,marge,abe,jacqueline]
```

It differs in treatment of free variables:
different instantiations lead to different answers:

```
?-  bagof(Y, ancestor(Y, X), L).
L =  [homer,marge,abe,jacqueline]
X = bart ? ;

L = [abe]
X = homer ? ...
```

Quantification

In the goal part of bagof/3, we can write

```
X^G
```

to hide (existentially quantify) X.

```
?-  bagof(Y, X^ancestor(Y, X), L).
L = [homer,bart,lisa,maggie,rod,
     todd,ralph,bart|...]
```

setof

setof/3 is like bagof/3, except it both sorts and eliminates duplicates.

```
| ?- bagof(Y,X^ancestor(X,Y),L).
L = [homer,bart,lisa,maggie,rod,
     todd,ralph,bart,lisa,maggie|...]

| ?- setof(Y,X^ancestor(X,Y),L).
L = [bart,homer,lisa,maggie,marge,
     patty,ralph,rod,selma,todd]
```

- So far, we have **statically** defined facts and rules, usually in a separate file.
- It is also possible to add and remove clauses **dynamically**.

assert/1

```
?- assert(p).
yes.

?- p.
yes

?- assert(q(1)).
yes.

?- q(X).
X = 1.
```

Fibonnacci, memoised

This can be useful when there is a lot of repeated computation.

```
:- dynamic memofib/2.
fib(N,K) :- memofib(N,K), !.

...
fib(N,K) :- N >= 2,
            M is N-1, fib(M,F),
            P is M-1, fib(P,G),
            K is F+G,
            assert(memofib(N,K)).
```

asserta.1, assertz/1

There is some control of where asserted statements appear in the clause order:

- `asserta/1` adds to the beginning of the KB
- `assertz/1` adds to the end of the KB

```
?- retract(p).
yes

?- p.
no.

?- retract(q(1)).
yes.

?- q(X).
no
```

dynamic predicates

- If you assert or retract an unused predicate interactively, Sicstus assumes it is dynamic.
- If you want assert/retract in programs, you need to **declare** the predicate as dynamic, as above for `memofib/2`.
- Generally a good idea to avoid assert/retract, unless you have good (efficiency) reason to use them.

Predicates to manipulate terms

can test to see if a term is a variable when called:

- `var(X)` holds
  `var(a)` does not hold
- Other tests, eg to see if term is atomic.

functor/3

This takes a term, and gives back the functor, and the arity (how many arguments).

```
?- functor(a,F,N).
F = a
N = 0

?- functor(f(a,b),F,N).
F = f
N = 2
```

arg/3

Given a number N and a compound term T, return the Nth argument to T:

```
?- arg(1,f(a,b),X).
X = a

?- arg(2,f(a,b),X).
X = b
```

=../2

The "universal" predicate =../2, that decomposes terms into their constituents as a list; works in both directions:

```
?- f(a,f(b,c)) =.. X.
X = [f,a,f(b,c)]

?- F =.. [g,a,f(b,c)].
F = g(a,f(b,c))
```

Together these predicates allow term manipulation, eg systematic generation.

Also:

- Further reading: LPN, chs 10, 11.
- Next session:
  - Parsing in Prolog
  - “Difference lists” for efficiency
  - Definite Clause Grammars (DCGs)