Computations and answers

Executing definite-clause definitions can sometimes lead to *non-termination* ("infinite loops") or even *unsound* behaviour (recall the idiosyncratic behaviour of $not/1$)

How are logic programs executed?

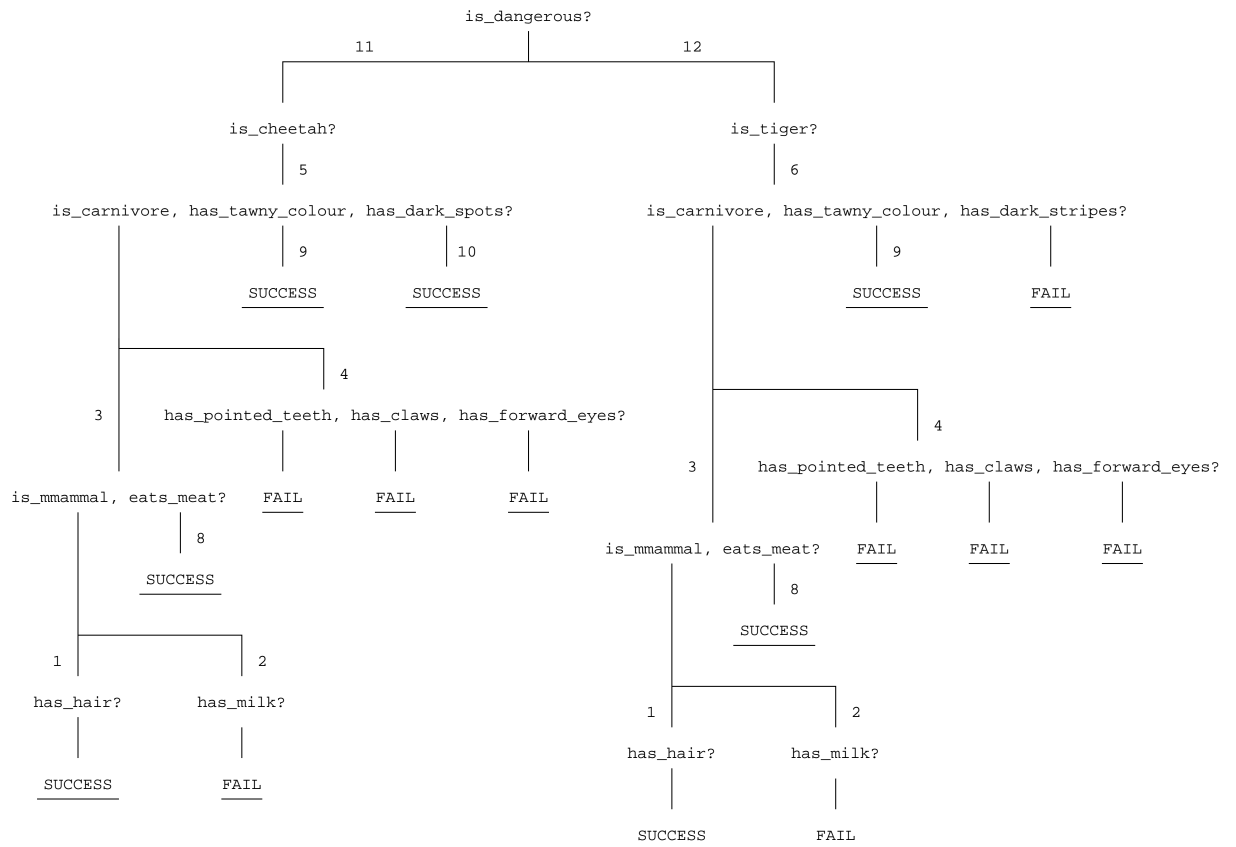
1. Execution of propositional logic programs
2. Execution of programs without recursion or negation
3. Execution of programs with recursion but no negation
4. Execution of programs with recursion and negation

Searching for answers: Proplog

Consider the following program:

1. $is_mammal \leftarrow has_hair$
2. $is_mammal \leftarrow has_milk$
3. $is_carnivore \leftarrow is_mammal, eats_meat$
4. $is_carnivore \leftarrow has_pointed_teeth, has_claws, has_forward_eyes$
5. $is_cheetah \leftarrow is_carnivore, has_tawny_colour, has_dark_spots$
6. $is_tiger \leftarrow is_carnivore, has_tawny_colour, has_dark_stripes$
7. $has_hair \leftarrow$
8. $eats_meat \leftarrow$
9. $has_tawny_colour \leftarrow$
10. $has_dark_spots \leftarrow$
11. $is_dangerous \leftarrow is_cheetah$
12. $is_dangerous \leftarrow is_tiger$

Search space for the query $is_dangerous$?



Even with this simple Proplog program, a number of choices have to be when searching this space to see if $is_dangerous$ is a logical consequence of clauses $1-12$

Research into *proof procedures* in logic programming has been concerned with

searching such spaces *efficiently* keeping in mind the properties of *soundness* or *completeness*

- Anything that is derived should be a logical consequence
- Any logical consequence should be derivable

Computation and Search rules

Typically, executing a logic program involves solving queries of the form: $l_1, l_2, \ldots, l_n$? where the l_i are literals

Two problems confront us when solving this query:

1. Which literal of the l_i should be solved first?
   - the rule governing this is called the *computation* rule
2. Which clause should be selected first, when more than one can be used to solve the literal selected?
   - the rule governing this is called the *search* rule

For a given program and query, the *computation* rule determines a tree of choices. The *search* rule determines the order in which this tree is searched (i.e. depth-first, breadth-first etc.)

Computation proceeds as follows. Given a program P and a query:
$l_1, l_2, \ldots, l_{j-1}, l_j, \ldots l_n?$

1. Use the computation rule to select l_j

2. Use the search rule to select a clause: $l_j \leftarrow b_1, b_2, \ldots b_k$ in P that can solve l_j. If none found, $STOP$

3. Solve the query:
   $l_1, l_2, \ldots, l_{j-1}, b_1, b_2, \ldots b_k \ldots l_n?$

– We will see later that the head of the clause selected does not have to

match *exactly* the literal selected. It will be enough if the two can *unify*, i.e. there is some substitution of variables for terms in the two literals that makes them the same

– The step of replacing the literal selected with the literals comprising the body of the clause (Step 3) will be seen as an application of the rule of inference known as *resolution*

Here is the earlier query with a computation rule that selects the leftmost literal first in the query:

The search rule will determine how this tree is searched for a leaf terminating in

$SUCCESS$ (for e.g. depth-first left-to-right, depth-first, right-to-left etc.)

- Different choices will affect efficiency, and sometimes even the ability to find the $SUCCESS$ leaf

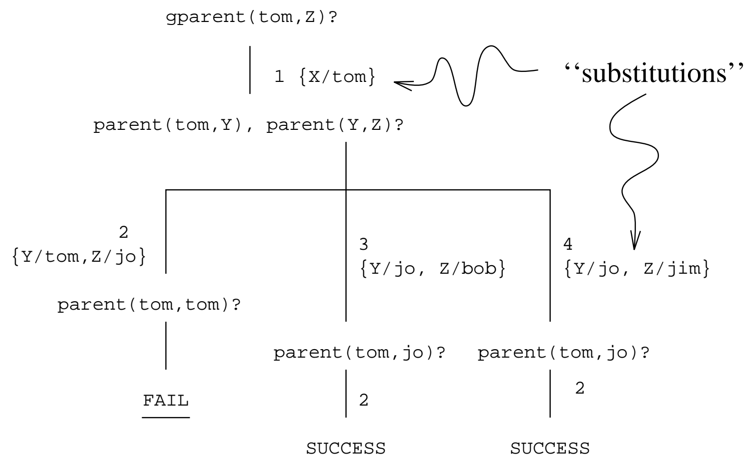
- Trees like this are known as *SLD-trees*: a reference to the trees obtained using a particular computation rule in conjunction with the inference rule of resolution for definite clause programs. More on this in *Proof theory*.

Datalog programs without recursion

We have looked at this earlier:

1. $gparent(X, Z) \leftarrow parent(X, Y), parent(Y, Z)$
2. $parent(tom, jo) \leftarrow$
3. $parent(jo, bob) \leftarrow$
4. $parent(jo, jim) \leftarrow$

Rightmost literal first computation rule:



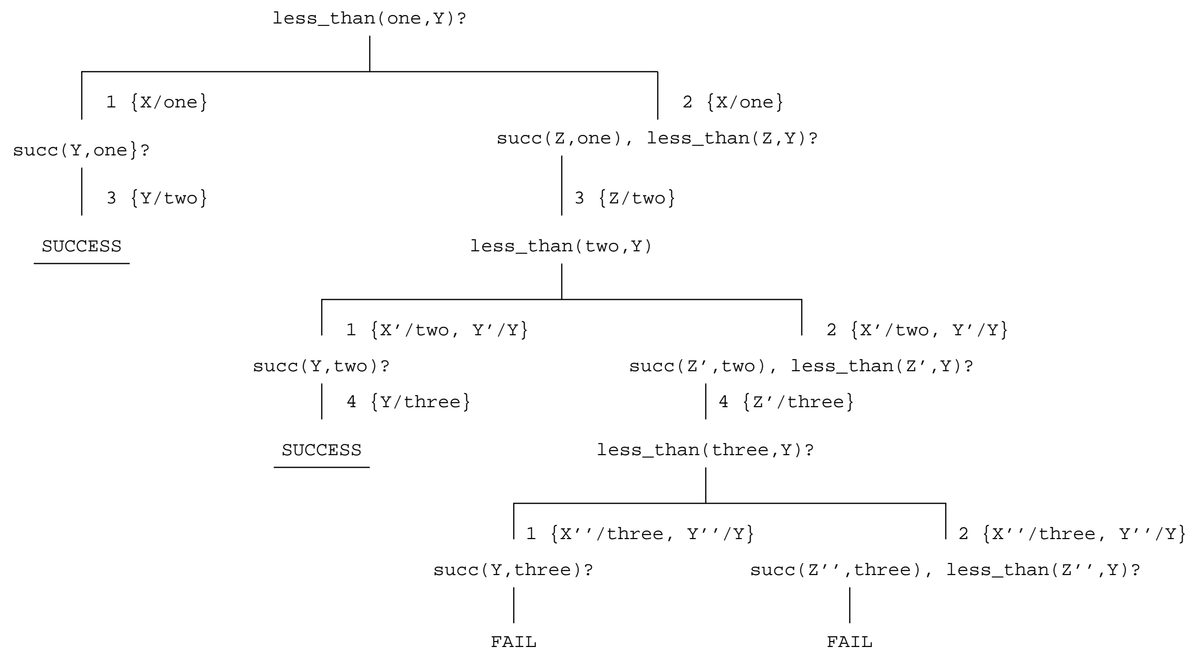
Datalog programs with recursion

Consider the following program:

1. $less_than(X,Y) \leftarrow succ(Y,X)$
2. $less_than(X,Y) \leftarrow succ(Z,Y), less_than(Z,Y)$
3. $succ(two,one) \leftarrow$
4. $succ(three,two) \leftarrow$

Leftmost literal rule for the query

less_than(one,Y)



Computation and search rules: completeness

One way to search the trees obtained so far is depth-first, left-to-right

– Since clauses that appear first (reading top to bottom) in the program have been drawn on the left, this search rule selects clauses in order of appearance in the program

Most logic programs are executed using the following:

Computation rule. Leftmost literal first

Search rule. Depth first search for clauses in order of appearance

Question. Will a logic-programming system with an arbitrary computation rule, and a depth-first search of clauses in some fixed order always find a leaf terminating in $SUCCESS$ (if one exists)?

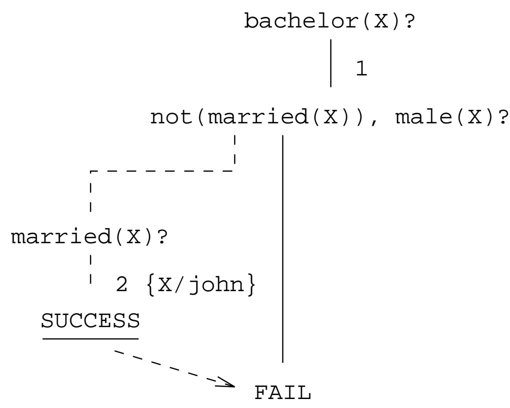
Answer. No

Finite failure: programs with negation

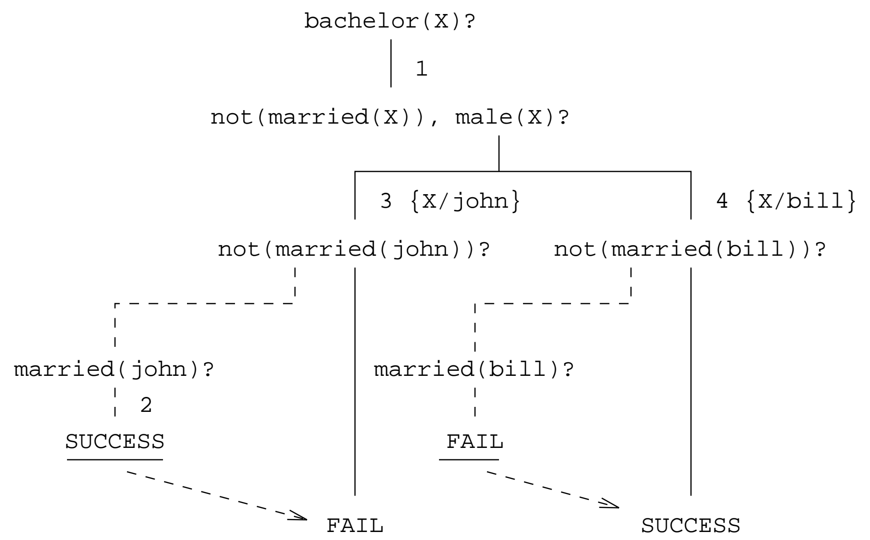
Consider the following program:

1. $bachelor(X) \leftarrow not(married(X)), male(X)$
2. $married(john) \leftarrow$
3. $male(john) \leftarrow$
4. $male(bill) \leftarrow$

Trees for the query $bachelor(X)$? with different computation rules are:



Leftmost literal computation rule



Rightmost literal computation rule

The query $not(q)$? succeeds iff q?
"finitely fails"

– Finitely failed tree: finite depth, finite depth and all computations end in $FAIL$

Summary

Given a program P, a computation rule R and a search rule S

- Think of all the ground queries that can be asked of P. For e.g. with the program with $less_than/2$ and $succ/2$ clauses, these are of the form:
  less_than(one,two)?, less_than(two,one)? ...,
  succ(one,two)? ...

- Such queries fall into 3 categories:

  1. Those which are answered yes because a $SUCCESS$ branch is found

  2. Those which are answered no because the query finitely fails

3. Those that are neither in categories 1 or 2