Evaluation of Recursive Queries

Part 1: Efficient fixpoint evaluation

"Seminaïve Evaluation"

Bottom-up evaluation

- Naïve

  Repeat

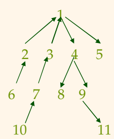
  Apply all rules

  Until no new tuples generated

- Seminaïve
  - If a rule is applied in iteration N, at least one body fact must be a fact generated in iteration N-1 (and not before!).
  - No application is repeated.

Example

up(2,1)	down(1,4)
up(3,1)	down(1,5)
up(6,2)	down(4,8)
up(7,3)	down(4,9)
up(10,7)	down(9,11)



r1: sg(X,Y) :- up(X,Z), down(Z,Y)

r2: sg(X,Y) :- up(X,Z_1), sg(Z_1,Z_2), down(Z_2,Y)

r3: sg(6,Y) ?

Naïve Evaluation proceed as follows:

Step(1)
sg(2,4), sg(2,5), sg(3,4), sg(3,5)

Step(2)
Iteration 1
sg(6,8), sg(6,9), sg(7,8), sg(7,9)

Iteration 2
sg(6,8), sg(6,9), sg(7,8), sg(7,9), sg(10,11)

Iteration 3
No new tuples

Seminaïve Evaluation proceed as follows:

Step(1)
sg(2,4), sg(2,5), sg(3,4), sg(3,5)

Step(2)
Iteration 1
sg(6,8), sg(6,9), sg(7,8), sg(7,9)

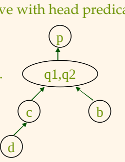
Iteration 2
sg(10,11)

Iteration 3
No new tuples

Notation

- Recursive Predicate
  - $p \rightarrow^* p$
- Mutually recursive predicate
  - $p \rightarrow^* q$, $q \rightarrow^* p$
- Strongly connected component (SCC)
  - A maximal set of mutually recursive predicates.
- Linear Rule
  - Only 1 body literal is mutually recursive with head predicate.

p(x) :- q1(x), q2(y).
q1(x) :- q2(x).
q2(x) :- q1(y), b(x,y).
q2(x) :- c(x,y).
c(x,y) :- d(x), d(y).



- Program Graph
  - Node = SCC
  - ARC : The 'Depends on' relation '→'

Seminaïve evaluation

- There are two components:
  - Rule Rewriting: Each rule in the program is replaced by a set of rules as follows:

$$p \text{ :- } \underbrace{p_1, p_2, \ldots, p_n}_{\text{recursive w.r.t. } p}, \underbrace{q_1, q_2, \ldots, q_m}_{\text{base predicates}}$$

is replaced by

$\delta p^{new}()$:- δp_1^{old}, $p_2, \ldots, p_n, q_1, q_2, \ldots, q_m$.

$\delta p^{new}()$:- p_1^{old}, δp_2^{old}, $\ldots, p_n, q_1, q_2, \ldots, q_m$.

…

$\delta p^{new}()$:- $p_1^{old}, p_2^{old}, \ldots, p_{n-1}^{old}$, δp_n^{old}, $q_1, q_2, \ldots, q_m$.

Special case: n=0, i.e. no recursive predicates

$\delta p^{new}()$:- $q_1, q_2, \ldots, q_m$.

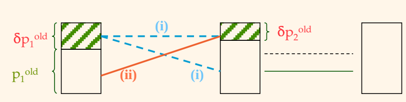
Example

$$p(x,z) \text{ :- } \underbrace{p_1(x,y), p_2(y,z)}_{\text{recursive w.r.t. } p}, \underbrace{q(z,w)}_{\text{base predicates}}$$

is replaced by

(i). $\delta p^{new}(x,z)$:- $\delta p_1^{old}(x,y)$, $p_2(y,z)$, q(z,w).

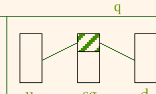
(ii). $\delta p^{new}(x,z)$:- $p_1^{old}(x,y)$, $\delta p_2^{old}(y,z)$, q(z,w).



Special case – 'Linear' rule:

$$sg(x,z) \text{ :- } \underbrace{u(x,v)}_{\text{base}}, sg(v,w), \underbrace{d(w,z)}_{\text{base}}.$$

$\delta sg^{new}(x,z)$:- u(x,v), $\delta sg^{old}(v,w)$, d(w,z).

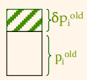


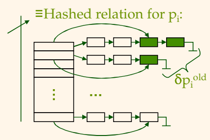
Seminaïve evaluation – Part 2

- Rule Evaluation
  - Repeatedly apply rule in 'iterations' until no new facts.
  - Iteration 1---Use all rules
  - Later iterations---Use only recursive rules
- In each iteration:
  - Apply rules
  - For each non-base predicate p, update associated relations as follows:

$$\left\{ \begin{array}{l} p_i^{old} = p_i^{old} \cup \delta p_i^{old} \\ \delta p_i^{old} = \delta p_i^{new} - p_i^{old} \\ p_i = p_i^{old} \cup \delta p_i^{old} \\ \delta p_i^{new} = \emptyset \end{array} \right.$$

after



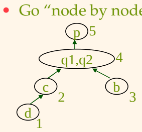


Initially: p_i, δp_i^{old} and δp_i^{new} = ∅

Seminaïve evaluation – Cont.

- Some observations
  - p_i — All known p_i facts.
  - δp_i^{old} — p_i facts (first) generated in previous iteration.
  - δp_i^{new} — p_i facts generated in this iteration.
  - p_i^{old} — $p_i - \delta p_i^{old}$ (∴ generated before prev. iteration)

  NO 'INFERENCE' is ever repeated!
- A refinement of rule evaluation:
  - Go "node by node, bottom-up" in program graph.



Evaluation order = 1, 2, 3, 4, 5

or 3, 1, 2, 4, 5

Top-down evaluation

- Given:

  Call: Q(?)

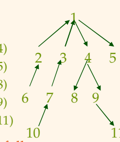
  Rule: Q IF $P_1 \wedge P_2 \ldots \wedge P_n$

  Generate subgoals:

  $P_1(?)\ P_2(?) \ldots P_n(?)$
- Advantage:
  - Computation is 'focused' in response to a query.
- Prolog is a language implemented in such a fashion.
  - Technique is called *resolution*

Example

up(2,1)	down(1,4)
up(3,1)	down(1,5)
up(6,2)	down(4,8)
up(7,3)	down(4,9)
up(10,7)	down(9,11)



r1: sg(X,Y) :- up(X,Z), down(Z,Y)

r2: sg(X,Y) :- up(X,Z_1), sg(Z_1,Z_2), down(Z_2,Y)

r3: sg(6,Y) ?

Prolog proceed as follows:

sg(6,y)_?
(r1)
    up(6,Z)_?(Z=2); down(2,Y)_? fails;
    up(6,Z)_? Fails on backtracking; (r1) fails.

(r2)
    up(6,Z_1)_?(Z1=2);
    sg(2,Z_2)_?
    (r1)
        up(2,Z′)_?(Z′=1); down(1,Y′)_?(Y′=Z_2=4);
    sg(2,Z_2)_? succeeds with Z_2 =4;
    down(4,Y)_?(Y=8);
sg(6,Y)_? succeeds with Y=8