Databases

http://www.dcs.fmph.uniba.sk/~plachetk

/TEACHING/DB2

http://www.dcs.fmph.uniba.sk/~sturc/databazy/rldb

Tomáš Plachetka, Ján Šturc

Faculty of mathematics, physics and informatics
Comenius University, Bratislava

Summer 2023–2024

Properties of the Magic Transformation

Magic Transformation is an optimization technique which **for a given query transforms a Datalog program to a target (magic) Datalog program** with the following properties:

- The target program **computes the same result** as the original program for the given query
- The (semi-) naïve computation of the target program **yields only the tuples computed by top-down computation** of the original program (QRGT)
- If the original program is **safe**, so is the target program

Magic Transformation

In this lecture, we will ignore negation in programs (in order keep things simple). We will focus on **recursive** programs with functional symbols

More on magic transformation in programs with negation:

Y. Chen: Magic sets and stratified databases, Int. J. Intelligent Systems, 1998

A. Behrend: Soft Stratification for Magic Set Based Query Evaluation in Deductive Databases, ACM PODS 2003

W. Faber, G. Greco, N. Leone: Magic Sets and their application to data integration, J. of Comp. and System Sciences, 2007

Magic and supplementary predicates

We begin with a realisable RGG:

• A **magic predicate** (unadorned) is assigned to each adorned IDB predicate, e.g. m_p_bf is assigned to p^{bf}. This magic predicate will be true only for tuples produced in some goal node p^{bf} by the top-down evaluation of the RGG. The variables of the magic predicate are the bound arguments in the goal calling p^{bf}

• A **supplementary predicate** is assigned to each rule node of the RGG (e.g. $sup_{2.1}$_bbf is assigned to $r_{2.1}$). The adornment of a supplementary predicate is inherited from its parent goal node. The bound variables of the node $r_{i.j}$ are the arguments of $sup_{i.j}$

In the sequel, we will sometimes omit adornments in the name of a magic or supplementary predicate when the adornments of the predicate are uniform in the RGG

Translation of an RGG to a magic program

Each **edge of the RGG** is **translated to a rule** of the magic program. We will distinguish between 5 classes of edges:

1. The edge from the query to the "first" IDB goal node of the RGG (seed)
2. All other edges pointing to an IDB goal node (recursive calls)
3. All edges entering a rule, i.e. pointing to a rule node $r_{i.0}$
4. All edges from a rule node $r_{i.j-1}$ to a rule node $r_{i.j}$, where $r_{i.j}$ is not the last node of the rule i
5. All edges from a rule node $r_{i.j-1}$ to a rule node $r_{i.j}$, where $r_{i.j}$ is the last node of the rule i

The remaining edges point to EDB predicates (database queries). These will be handled together with edges in classes 4 and 5

Class 1: Query → IDB goal node

?- q(B_1, …, B_m, F_1, …, F_n).

where B_1, …, B_m are bound arguments of q initialises the magic predicate for (adorned) q:

m_q(B_1, …, B_m).

Relational algebra:

m_q := $\pi_{B1, \ldots, Bm}$ (q)

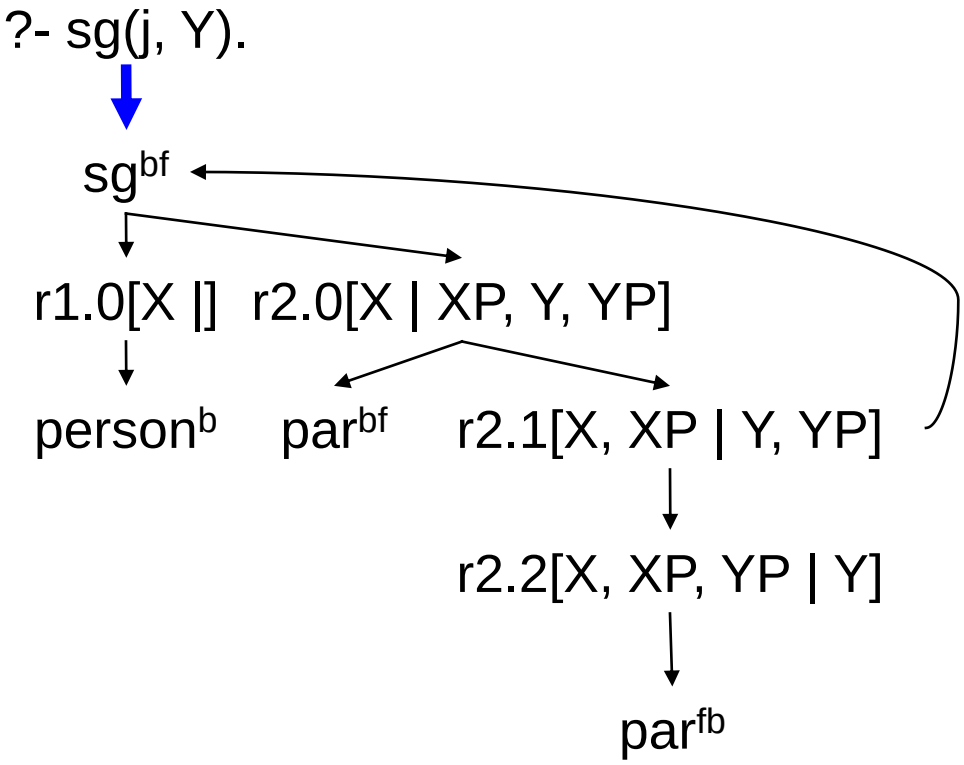
Class 1: Query → IDB goal node: Example

Example:

r1: sg(X, X) ← person(X).

r2: sg(X, Y) ← par(X, XP), sg(XP, YP), par(Y, YP).

?- sg(j, Y).



Magic program:

m_sg_bf(j).

Class 2: Non-query node → IDB goal node

r_i: $q(\ldots) \leftarrow G_1, \ldots G_{j-1}, p(B_1, \ldots, B_m, F_1, \ldots, F_n), \ldots, G_l$.

where $G_j = p(B_1, \ldots, B_m, F_1, \ldots, F_n)$ with bound arguments $B_1, \ldots, B_m$ updates the (unadorned) magic predicate for the (adorned) predicate p (the arguments of m_p are as they appear in the call):

$m\_p(B_1, \ldots, B_m) \leftarrow sup_{i.j-1}(B_1, \ldots, B_m, \ldots)$.

Relational algebra:

$m\_p := atov(G_j, sup_{i.j-1}(\ldots))$

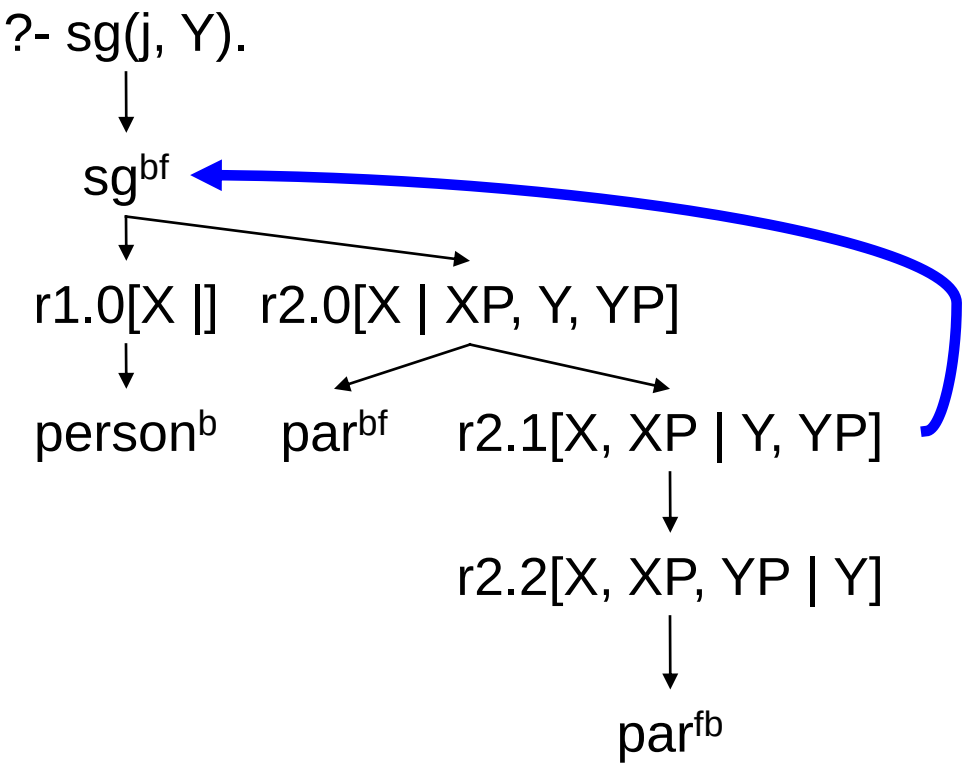
Class 2: Non-query node → IDB goal node: Example

Example:

r1: sg(X, X) ← person(X).

r2: sg(X, Y) ← par(X, XP), **sg(XP, YP)**, par(Y, YP).

?- sg(j, Y).



Magic program:

m_sg_bf(XP) ←
sup2.1_bfbf(X, XP).

Class 3: IDB goal node → First rule node

r_i: $q(B_1, \ldots, B_m, F_1, \ldots, F_n) \leftarrow G_1, \ldots G_{j-1}, \ldots, G_l$.

where $b_1, \ldots, b_m$ are bound arguments of (adorned) q initialises the supplementary predicate of the rule r_i with the magic predicate m_q (with the bound arguments as they appear in the head):

$sup_{i.0}(B_1, \ldots, B_m) \leftarrow m\_q(B_1, \ldots, B_m)$.

Relational algebra:

$sup_{i.0} := atov(q(B_1, \ldots, B_m, F_1, \ldots, F_n), m\_q)$

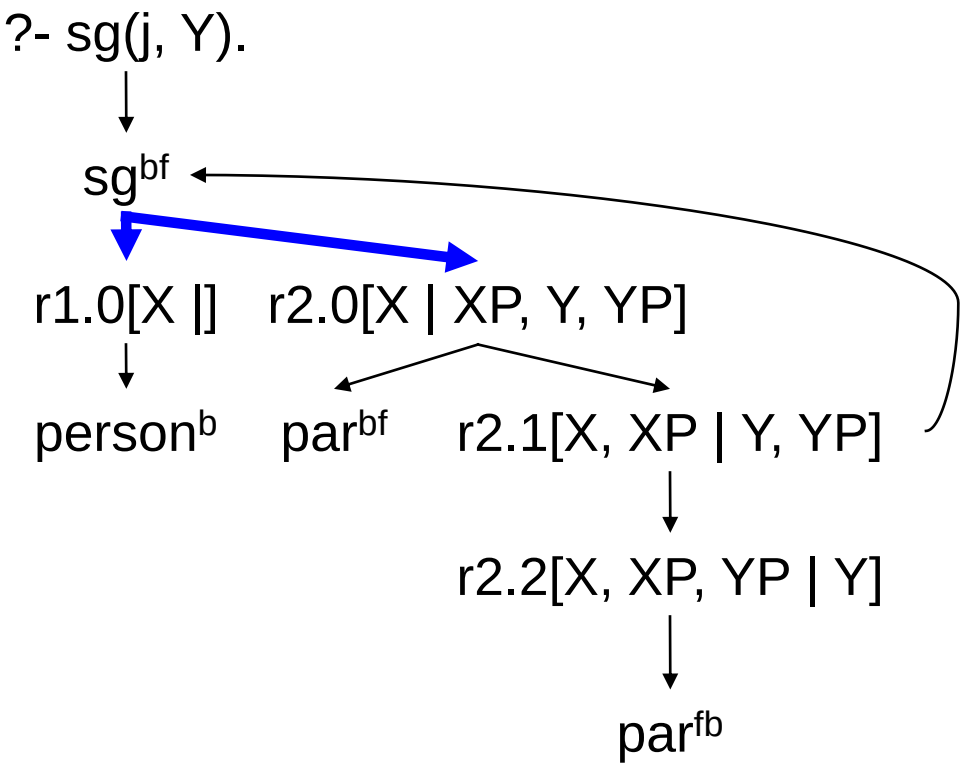
Class 3: IDB goal node → First rule node: Example

Example:

r1: sg(X, X) ← person(X).

r2: sg(X, Y) ← par(X, XP), sg(XP, YP), par(Y, YP).

?- sg(j, Y).



Magic program:

sup1.0_b(X) ←
m_sg_bf(X).
sup2.0_bfff(X) ←
m_sg_bf(X).

Class 4: Rule node → Next rule node

r_i: $q(B_1, \ldots, B_m, F_1, \ldots, F_n) \leftarrow G_1, \ldots G_{j-1}, G_j, G_{j+1}, \ldots, G_l$.

The supplementary predicate $sup_{i.j+1}$ is defined by the previous supplementary predicate $sup_{i,j}$ and the goal G_j:

$sup_{i.j+1}(\ldots) \leftarrow sup_{i.j}(\ldots), G_j$.

Relational algebra:

$sup_{i.j+1} := sup_{i.j} \bowtie G_j$

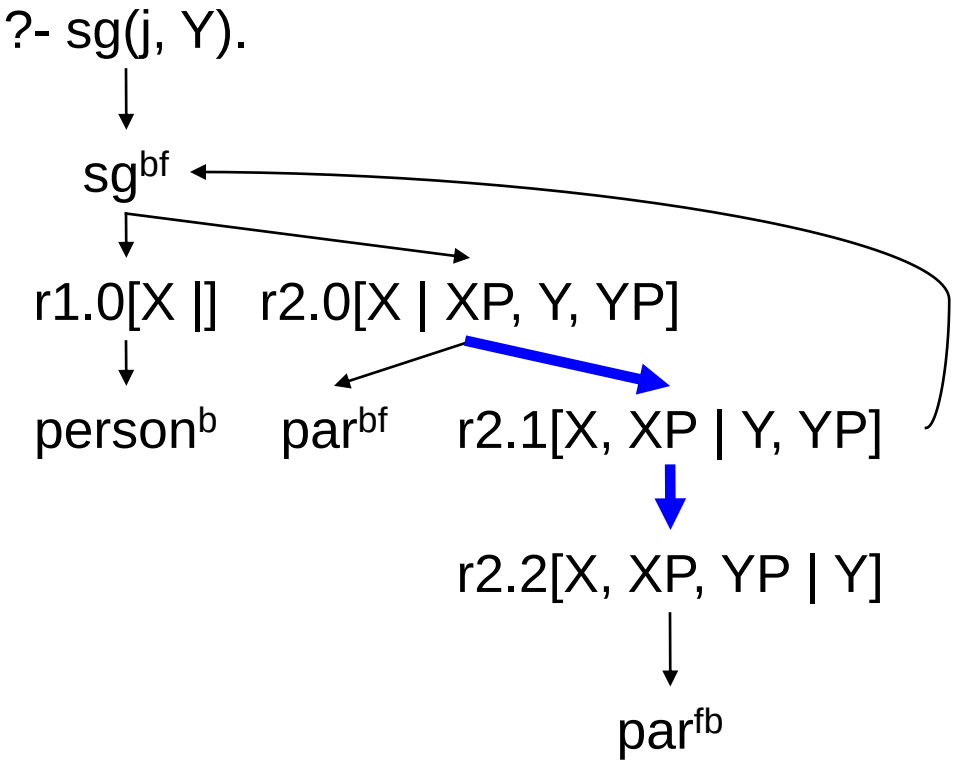
Class 4: Rule node → Next rule node: Example

Example:

r1: sg(X, X) ← person(X).

r2: sg(X, Y) ← par(X, XP), sg(XP, YP), par(Y, YP).

?- sg(j, Y).



Magic program:

sup2.1_bfbf(X, XP) ←
sup2.0_bfff(X),
par(X, XP).

sup2.2_bfbb(X, XP, YP) ←
sup2.1_bfbf(X, XP),
sg(XP, YP).

Class 5: Rule node → Last rule node

r_i: $q(B_1, \ldots, B_m, F_1, \ldots, F_n) \leftarrow G_1, \ldots G_l$.

The head of the rule r_i is defined by the last supplementary predicate sup_{l-1} and the goal G_l:

$q(B_1, \ldots, B_m, F_1, \ldots, F_n) \leftarrow sup_{i.l-1}(\ldots), G_l$.

Relational algebra:

$q := vtoa(q(B_1, \ldots, B_m, F_1, \ldots, F_n), sup_{i.l-1} \bowtie G_l)$

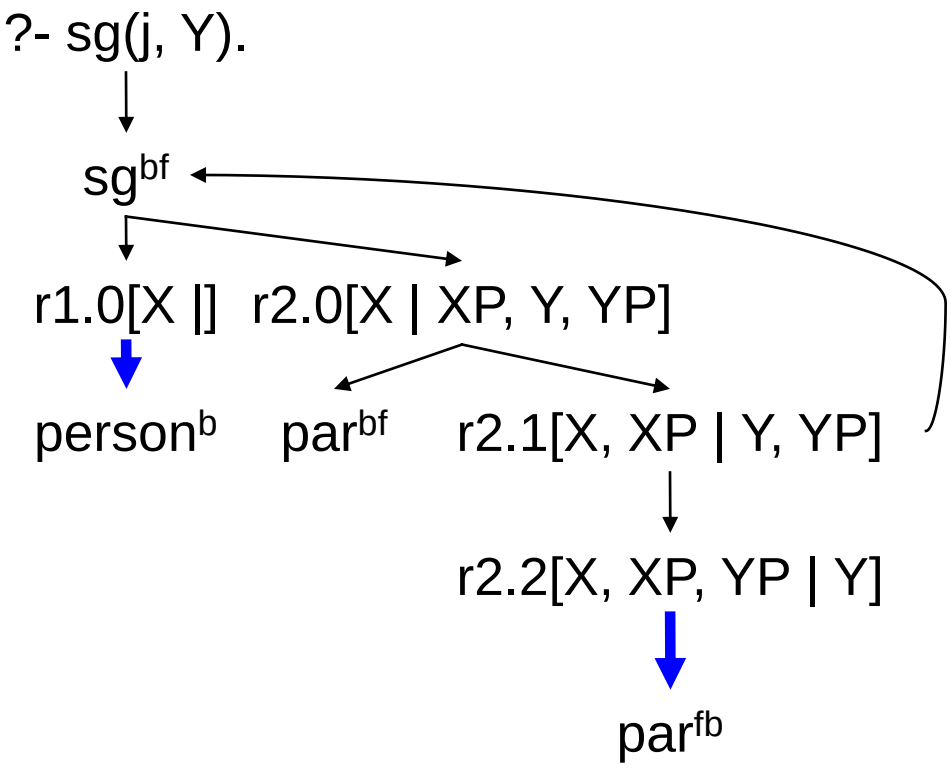
Class 5: Rule node → Last rule node: Example

Example:

r1: sg(X, X) ← person(X).

r2: sg(X, Y) ← par(X, XP), sg(XP, YP), par(Y, YP).

?- sg(j, Y).



Magic program:

sg(X, X) ←
sup1.0_b(X),
person(X).

sg(X, Y) ←
sup2.2_bfbb(X, XP, YP),
par(Y, YP).

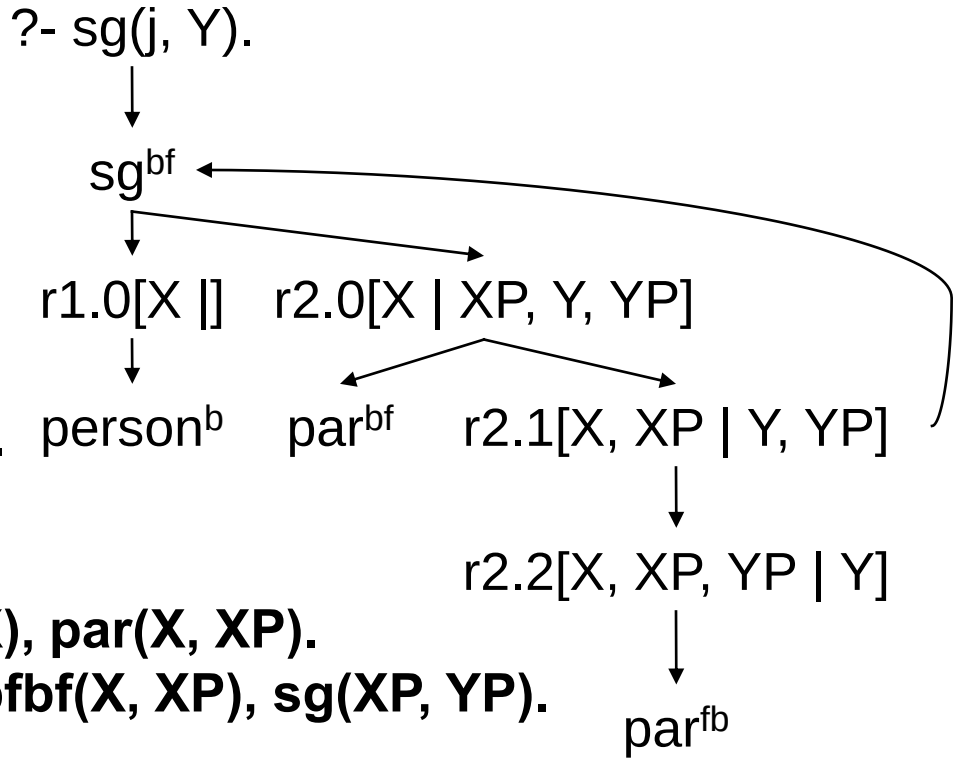
Example, summary

Example:

r1: sg(X, X) ← person(X).

r2: sg(X, Y) ← par(X, XP), sg(XP, YP), par(Y, YP).

?- sg(j, Y).



Magic program:

m_sg_bf(j).

m_sg_bf(XP) ← sup2.1_bfbf(X, XP).

sup1.0_b(X) ← m_sg_bf(X).

sup2.0_bfff(X) ← m_sg_bf(X).

sup2.1_bfbf(X, XP) ← sup2.0_bfff(X), par(X, XP).

sup2.2_bfbb(X, XP, YP) ← sup2.1_bfbf(X, XP), sg(XP, YP).

sg(X, X) ← sup1.0_b(X), person(X).

sg(X, Y) ← sup2.2_bfbb(X, XP, YP), par(Y, YP).

?- sg(j, Y).

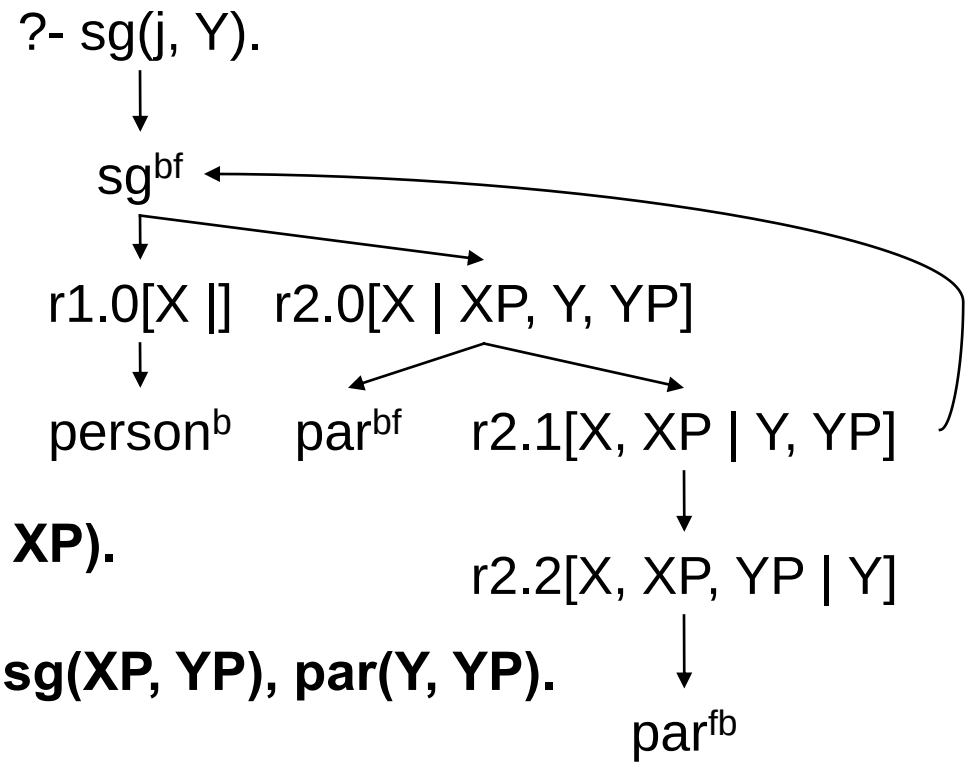
Example, summary

Example:

r1: sg(X, X) ← person(X).

r2: sg(X, Y) ← par(X, XP), sg(XP, YP), par(Y, YP).

?- sg(j, Y).



Magic program simplified (sup predicates are replaced with their definitions):

m_sg_bf(j).

m_sg_bf(XP) ← m_sg_bf(X), par(X, XP).

sg(X, X) ← m_sg_bf(X), person(X).

sg(X, Y) ← m_sg_bf(X), par(X, XP), sg(XP, YP), par(Y, YP).

?- sg(j, Y).

Example: path lengths (a built-in predicate)

r1: p(X, Y, D) ← a(X, Y, D).

r2: p(X, Y, D) ← p(X, Z, E), p(Z, Y, F), **sum(E, F, D)**.

?- p(x0, Y, D).

/* Class 1 */

m_p(x0).

/* Class 2 */

m_p(X) ← sup2.0(X).

m_p(Z) ← sup2.1(X, Z, E).

/* Class 3 */

sup1.0(X) ← m_p(X).

sup2.0(X) ← m_p(X).

/* Class 4 */

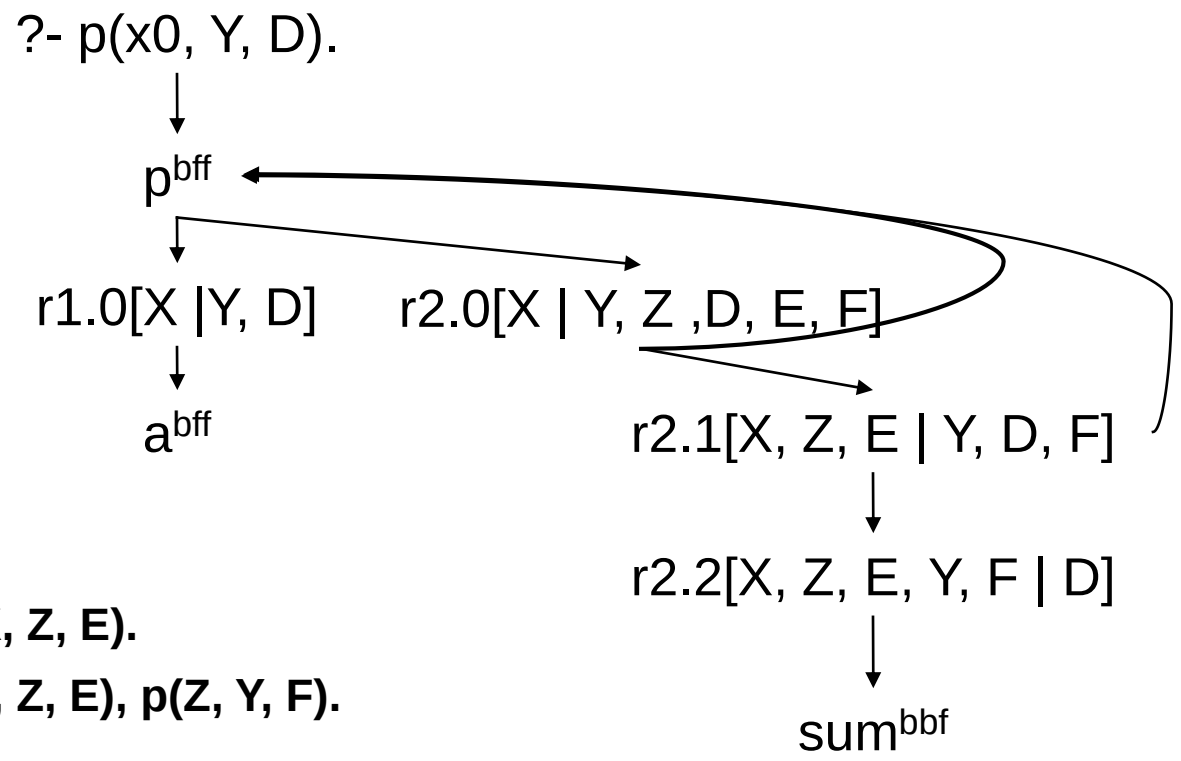
sup2.1(X, Z, E) ← sup2.0(X), p(X, Z, E).

sup2.2(X, Z, E, Y, F) ← sup2.1(X, Z, E), p(Z, Y, F).

/* Class 5 */

p(X, Y, D) ← sup1.0(X), a(X, Y, D).

p(X, Y, D) ← sup2.2(X, Z, E, Y, F), sum(E, F, D).



Example: path lengths (a built-in predicate)

r1: p(X, Y, D) ← a(X, Y, D).

r2: p(X, Y, D) ← p(X, Z, E), p(Z, Y, F), **sum(E, F, D)**.

?- p(x0, Y, D).

Magic program, simplified:

m_p(x0).

m_p(Z) ←

m_p(X),

p(X, Z, _).

p(X, Y, D) ←

m_p(X),

a(X, Y, D).

p(X, Y, D) ←

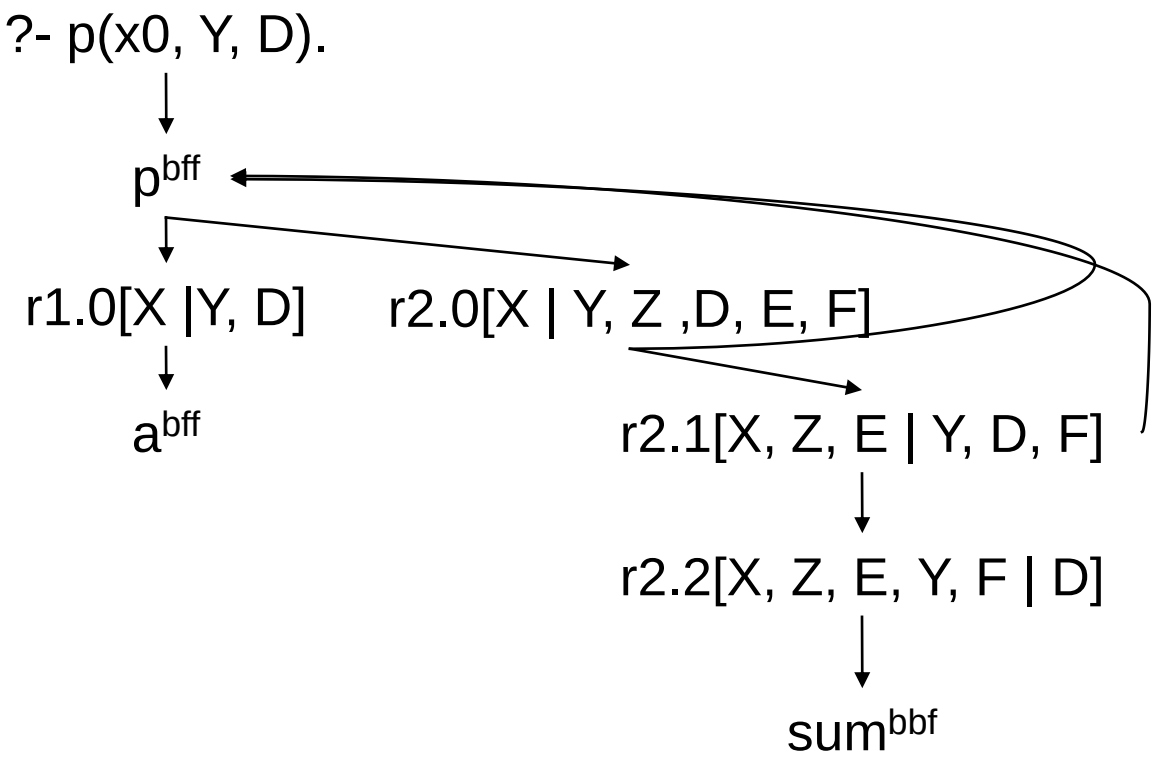
m_p(X),

p(X, Z, E),

p(Z, Y, F),

sum(E, F, D).

?- p(x0, Y, D).



Example: underpaid IT employees (subtotal, built-in <)

Underpaid IT employees earn less than the average salary in their department

EDB: emp(Name, Job, Department, Salary)

r1: upaid(N, J) ← emp(N, J, D, S), **subtotal(emp(_, _, D, S), [D], [A = avg(S)])**, S **<** A.

?- upaid(N, it).

Magic program:

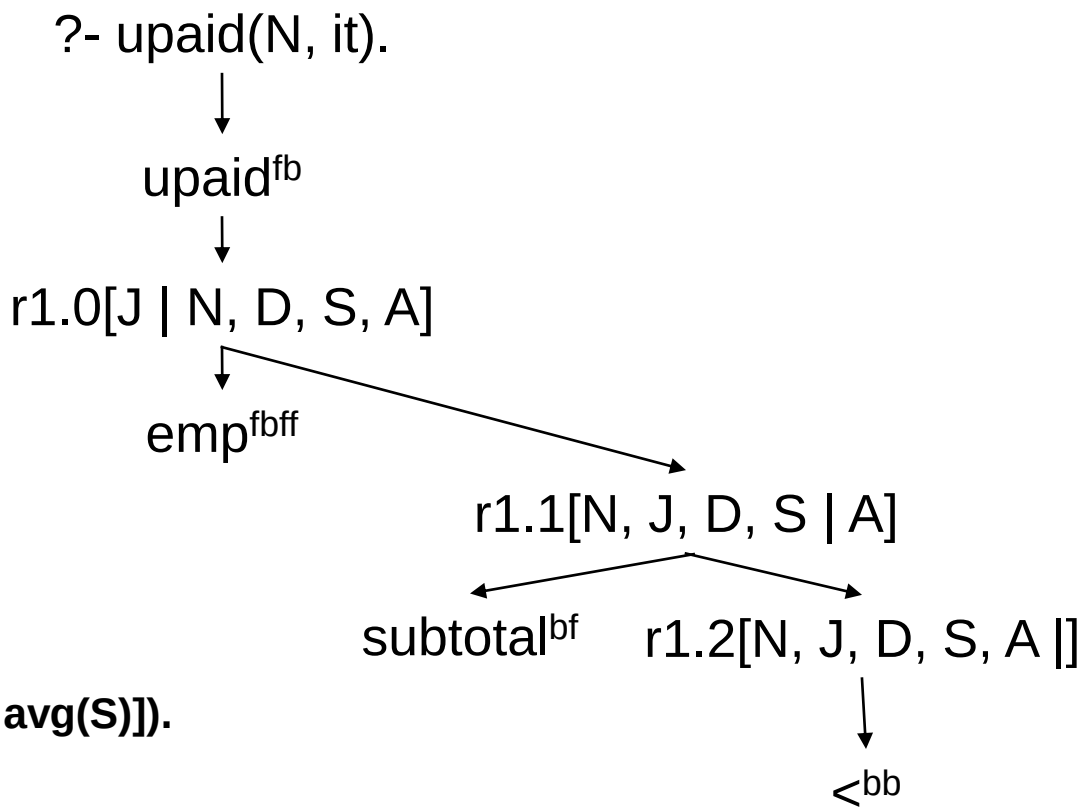
m_upaid(it).

sup1.0(J) ←
    m_upaid(J).

sup1.1(N, J, D, S) ←
    sup1.0(J),
    emp(N, J, D, S).

sup1.2(N, J, D, S, A) ←
    sup1.1(N, J, D, S),
    subtotal(emp(_, _, D, S), [D], [A = avg(S)]).

upaid(N, J) ←
    sup1.2(N, J, D, S, A),
    S < A.



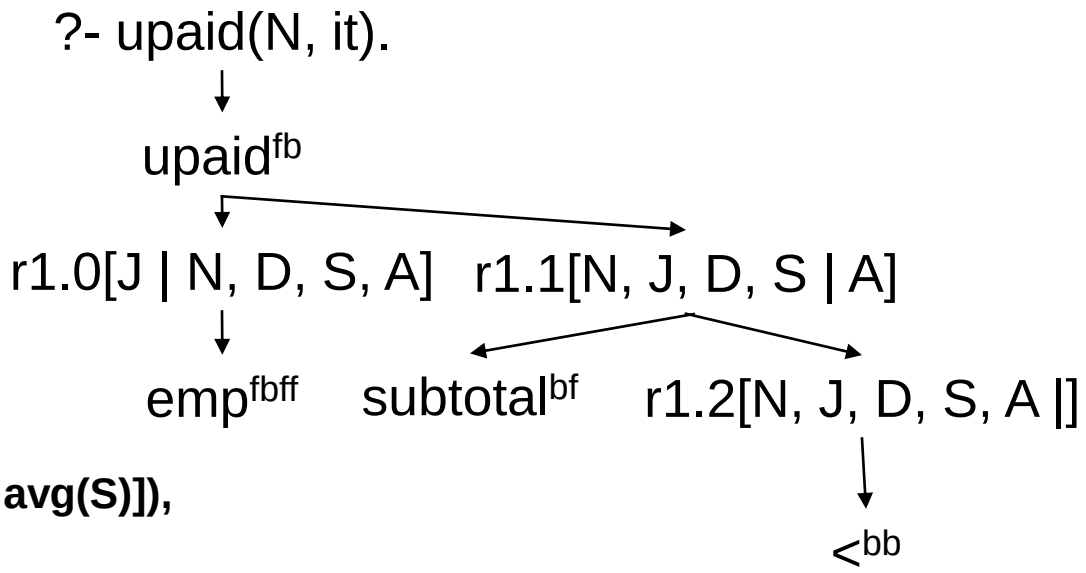
Example: underpaid IT employees (subtotal, built-in <)

Underpaid IT employees earn less than the average salary in their department

EDB: emp(Name, Job, Department, Salary)

r1: upaid(N, J) ← emp(N, J, D, S), **subtotal(emp(_, _, D, S), [D], [A = avg(S)])**, S **<** A.

?- upaid(N, it).



Magic program, simplified:

m_upaid(it).

upaid(N, J) ←

m_upaid(J),

emp(N, J, D, S),

subtotal(emp(_, _, D, S), [D], [A = avg(S)]),

S < A.

Magic program, yet more simplified:

upaid(N, it) ←

emp(N, it, D, S),

subtotal(emp(_, _, D, S), [D], [A = avg(S)]),

S < A.

Example: syntactical analysis (functional symbols)

This query verifies whether **x * (y + z)** is an expression, for EDB = {ident(.) = {x, y, z}}

?- expr(mult(x, bra(plus(y, z)))).

r1: expr(T) ← term(T).

r2: expr(plus(T, E)) ← term(T), expr(E).

r3: term(F) ← factor(F).

r4: term(mult(T, F)) ← term(T), factor(F).

r5: factor(F) ← ident(F).

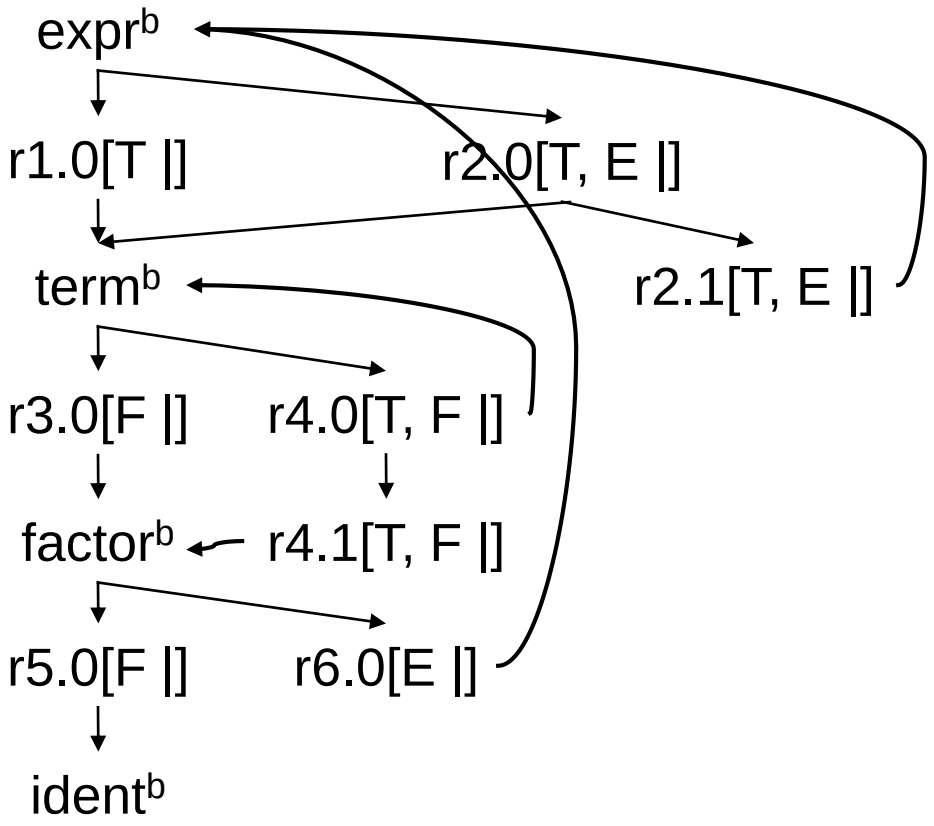
r6: factor(bra(E)) ← expr(E).

A BNF grammar:

<expr> ::= <term> | <term> "+" <expression>

<term> ::= <factor> | <term> "*" <factor>

<factor> ::= <ident> | "(" <expression> ")"



Example: syntactical analysis (functional symbols)

?- expr(mult(x, bra(plus(y, z)))).

r1: expr(T) ← term(T).

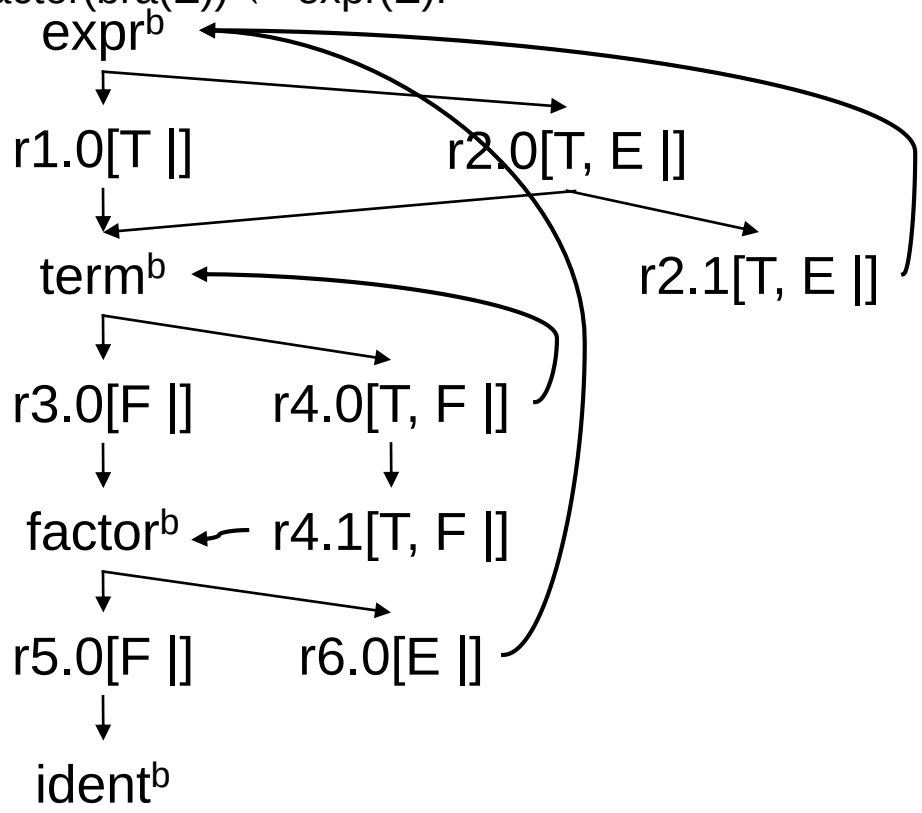
r2: expr(plus(T, E)) ← term(T), expr(E).

r3: term(F) ← factor(F).

r4: term(mult(T, F)) ← term(T), factor(F).

r5: factor(F) ← ident(F).

r6: factor(bra(E)) ← expr(E).



Magic program:

?- expr(mult(x, bra(plus(y, z)))).

m_expr(mult(x, bra(plus(y, z)))).

m_expr(E) ← sup2.1(T, E).

m_expr(E) ← sup6.0(E).

m_term(T) ← sup1.0(T).

m_term(T) ← sup2.0(T, E).

m_term(T) ← sup4.0(T, F).

m_factor(F) ← sup3.0(F).

m_factor(F) ← sup4.1(T, F).

sup1.0(T) ← m_expr(T).

sup2.0(T, E) ← m_expr(plus(T, E)).

sup3.0(F) ← m_term(F).

sup4.0(T, F) ← m_term(mult(T, F)).

sup5.0(F) ← m_factor(F).

sup6.0(E) ← m_factor(bra(E)).

sup2.1(T, E) ← sup2.0(T, E), term(T).

sup4.1(T, F) ← sup4.0(T, F), term(T).

expr(T) ← sup1.0(T), term(T).

expr(plus(T, E)) ← sup2.1(T, E), expr(E).

term(F) ← sup3.0(F), factor(F).

term(mult(T, F)) ← sup4.1(T, F), factor(F).

factor(F) ← sup5.0(F), ident(F).

factor(bra(E)) ← sup6.0(E), expr(E).

Example: syntactical analysis (functional symbols)

?- expr(mult(x, bra(plus(y, z)))).

r1: expr(T) ← term(T).

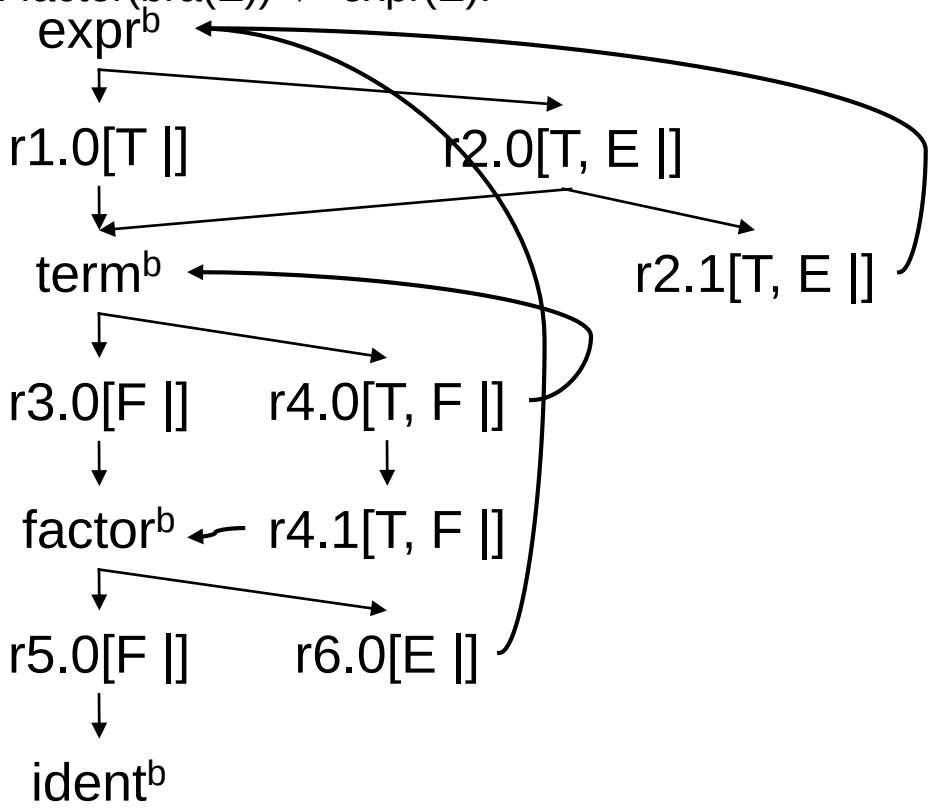
r2: expr(plus(T, E)) ← term(T), expr(E).

r3: term(F) ← factor(F).

r4: term(mult(T, F)) ← term(T), factor(F).

r5: factor(F) ← ident(F).

r6: factor(bra(E)) ← expr(E).



Magic program, simplified:

?- expr(mult(x, bra(plus(y, z)))).

m_expr(mult(x, bra(plus(y, z)))).

m_expr(E) ← m_expr(plus(T, E)), term(T).

m_expr(E) ← m_factor(bra(E)).

m_term(T) ← m_expr(T).

m_term(T) ← m_expr(plus(T, _)).

m_term(T) ← m_term(mult(T, _)).

m_factor(F) ← m_term(F).

m_factor(F) ← m_term(mult(T, F)), term(T).

expr(T) ← m_expr(T), term(T).

expr(plus(T, E)) ← m_expr(plus(T, E)), term(T), expr(E).

term(F) ← m_term(F), factor(F).

term(mult(T, F)) ← m_term(mult(T, F)), term(T), factor(F).

factor(F) ← m_factor(F), ident(F).

factor(bra(E)) ← m_factor(bra(E)), expr(E).

Example: syntactical analysis (functional symbols)

Magic program, simplified:

```
?- expr(mult(x, bra(plus(y, z)))).
m_expr(mult(x, bra(plus(y, z)))).
m_expr(E) ← m_expr(plus(T, E)), term(T).
m_expr(E) ← m_factor(bra(E)).
m_term(T) ← m_expr(T).
m_term(T) ← m_expr(plus(T, _)).
m_term(T) ← m_term(mult(T, _)).
m_factor(F) ← m_term(F).
m_factor(F) ← m_term(mult(T, F)), term(T).
expr(T) ← m_expr(T), term(T).
expr(plus(T, E)) ← m_expr(plus(T, E)), term(T), expr(E).
term(F) ← m_term(F), factor(F).
term(mult(T, F)) ← m_term(mult(T, F)), term(T), factor(F).
factor(F) ← m_factor(F), ident(F).
factor(bra(E)) ← m_factor(bra(E)), expr(E).
```

In general, magic predicates serve as filters. In this case, only sub-expressions of the query expression enter the computation

There still is a slight inefficiency in comparison with a hand-made compiler (e.g. in C): m_term and m_factor contain also expressions which are neither terms nor factors

The bottom-up computation of the query with the original program does not terminate. The bottom-up computation with the magic program terminates

Example: one-way paths (negation)

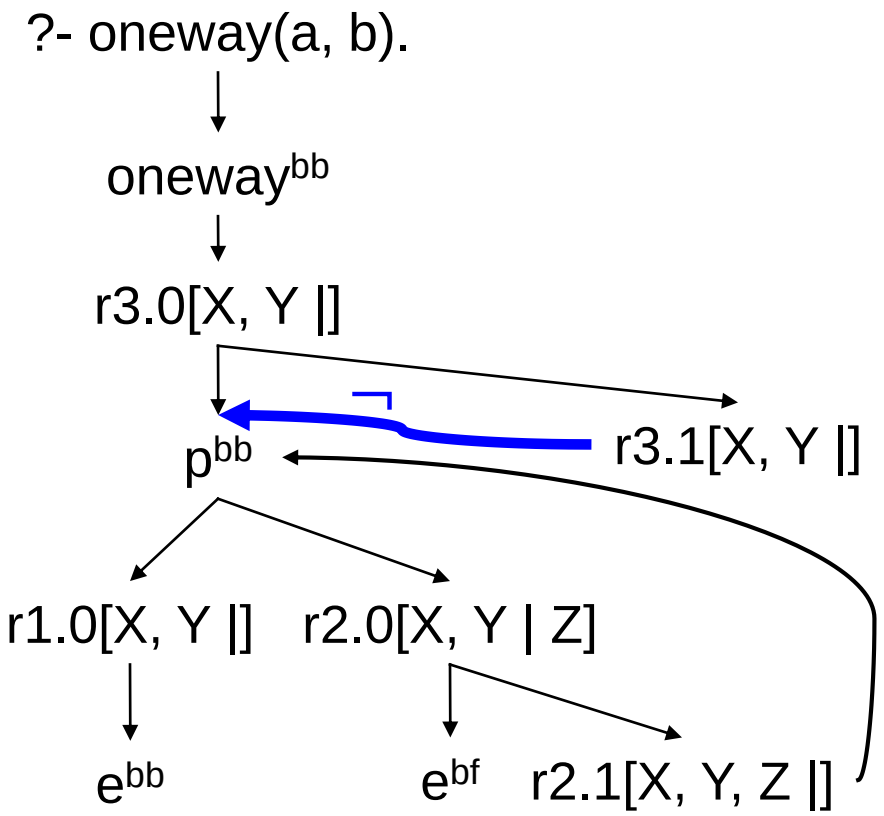
r1: p(X, Y) ← e(X, Y).

r2: p(X, Y) ← e(X, Z), p(Z, Y).

r3: oneway(X, Y) ← p(X, Y), ¬ p(Y, X).

?- oneway(a, b).

Note that **this program is stratified**



Example: one-way paths (negation)

Magic program:

?- oneway(a, b).

m_oneway(a, b).

m_p(Z, Y) ← sup2.1(X, Y, Z).

m_p(X, Y) ← sup3.0(X, Y).

m_p(Y, X) ← sup3.0(X, Y), ¬ sup3.1(X, Y).

sup1.0(X, Y) ← m_p(X, Y).

sup2.0(X, Y) ← m_p(X, Y).

sup3.0(X, Y) ← m_oneway(X, Y).

sup2.1(X, Y, Z) ← sup2.0(X, Y), e(X, Z).

sup3.1(X, Y) ← sup3.0(X, Y), p(X, Y).

p(X, Y) ← sup1.0(X, Y), e(X, Y).

p(X, Y) ← sup2.1(X, Y, Z), p(Z, Y).

oneway(X, Y) ← sup3.1(X, Y), **¬ p(Y, X)**.

Simplified magic program:

?- oneway(a, b).

m_oneway(a, b).

m_p(Z, Y) ← m_p(X, Y), e(X, Z).

m_p(X, Y) ← m_oneway(X, Y).

m_p(Y, X) ← m_oneway(X, Y),

¬ p(X, Y).

p(X, Y) ← m_p(X, Y), e(X, Y).

p(X, Y) ← m_p(X, Y), e(X, Z), p(Z, Y).

oneway(X, Y) ← m_oneway(X, Y),

p(X, Y), ¬ p(Y, X).

Negation requires a special treatment not only in the magic transformation, but **also during the computation of the magic program**

Expansion of IDB goals

For some programs (with a “simple recursion pattern”), the magic transformation can be replaced by the technique of expansion of IDB goals

Example:

p(X, Y) ← q(X, Y, Y, a).
q(A, B, C, D) ← p(A, B), p(C, D).
q(A, B, C, D) ← b(A, B, C, D).
?- p(X, c).

The first rule is the only one which allows for proving p(X, c). We can equivalently rewrite the program by substituting p(., .) with its definition in both the program and the query

Expansion of IDB goals

The reformulated program:
q(A, B, C, D) ← b(A, B, C, D).
q(A, B, C, D) ← q(A, B, B, a), q(C, D, D, a).
?- q(X, c, c, a).

Expand the two rules by unifying the query with the heads:
q(X, c, c, a) ← b(X, c, c, a).
q(X, c, c, a) ← q(X, c, c, a), q(c, a, a, a).

The first subgoal of the second rule is identical with the head of the rule, which makes this rule useless for derivation of q(X, c, c, a)

The program with the query simplifies to a simple (non-recursive) database query:
?- b(X, c, c, a).

Generalised Magic Transformation

The magic transformation can be generalised in order to **use all kinds of bindings in the rule bodies**, including constants and duplicated variables

The construction of a generalised magic program is simpler than the construction of an ordinary magic program, as it needs **no rectification and no adornments** (of course, adornments are still important for built-in predicates and EDB queries)

The computation of a generalised magic program is hard. It can no longer be computed using an "ordinary relational algebra", the resulting program is not necessarily safe. The computation involves **working with general terms in the relational algebra** (e.g. storing terms with variables into a database, unification joins and antijoins, ...)

Generalised Magic Transformation

Construction of a generalised magic program:

- Each IDB predicate is assigned one magic predicate (ignoring adornments), with as many arguments as in the IDB predicate
- Each supplementary predicate has as many arguments as the number of variables in the rule

The rest is similar to the construction of an ordinary magic program

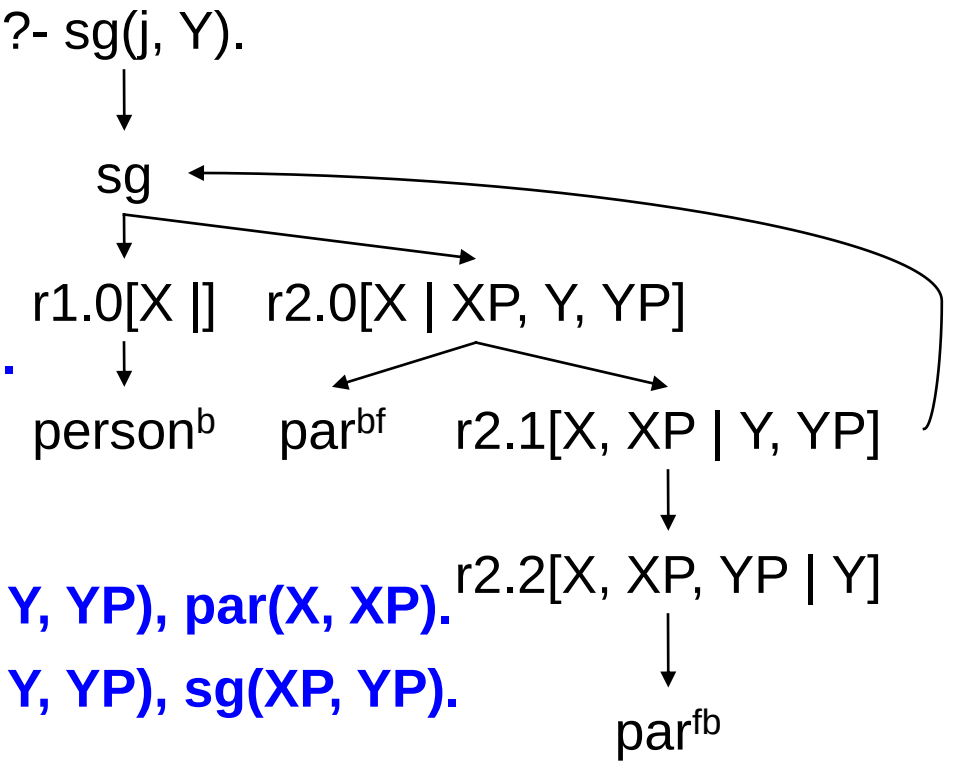
Generalised Magic Transformation

Example:

r1: sg(X, X) ← person(X).

r2: sg(X, Y) ← par(X, XP), sg(XP, YP), par(Y, YP).

?- sg(j, Y).



Generalised magic program:

m_sg(j, Y).

m_sg(XP, YP) ← sup2.1(X, XP, Y, YP).

sup1.0(X) ← m_sg(X, X).

sup2.0(X, XP, Y, YP) ← m_sg(X, Y).

sup2.1(X, XP, Y, YP) ← sup2.0(X, XP, Y, YP), par(X, XP).

sup2.2(X, XP, Y, YP) ← sup2.1(X, XP, Y, YP), sg(XP, YP).

sg(X, X) ← sup1.0(X), person(X).

sg(X, Y) ← sup2.2(X, XP, Y, YP), par(Y, YP).

Generalised Magic Transformation

Example:

r1: sg(X, X) ← person(X).

r2: sg(X, Y) ← par(X, XP), sg(XP, YP), par(Y, YP).

?- sg(j, Y).

Generalised magic program, simplified:

m_sg(j, _).

m_sg(XP, YP) ← m_sg(X, _), par(X, XP).

sg(X, X) ← m_sg(X, X), person(X).

sg(X, Y) ← m_sg(X, Y), par(X, XP), sg(XP, YP), par(Y, YP).