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

Datalog vs Prolog: bottom-up vs top-down

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

Datalog vs Prolog: bottom-up vs top-down

No difference in syntax, a **world of difference in computation**

Examples:

- **Simple join:** q(X, Y, Z) ← r(X, Y), s(Y, Z). ?- q(a, b, c).

Bottom-up (Datalog's naive evaluation) computes the join, then the selection:

$B(X, Y, Z) = R(X, Y) \bowtie S(Y, Z)$; $Q(X, Y, Z) = \sigma_{X=a \land Y=b \land Z=c} B(X, Y, Z)$

Top-down (Prolog's SLD resolution) starts with the query. It attempts to prove q(a, b, c) by finding the rule whose head unifies with q(a, b, c) and finding instances of variables which satisfy all subgoals in the body of that rule

- **Recursion, a graph path:**

p(X, Y) ← e(X, Y).   p(X, Y) ← e(X, Z), p(Z, Y). ?- p(a, b).

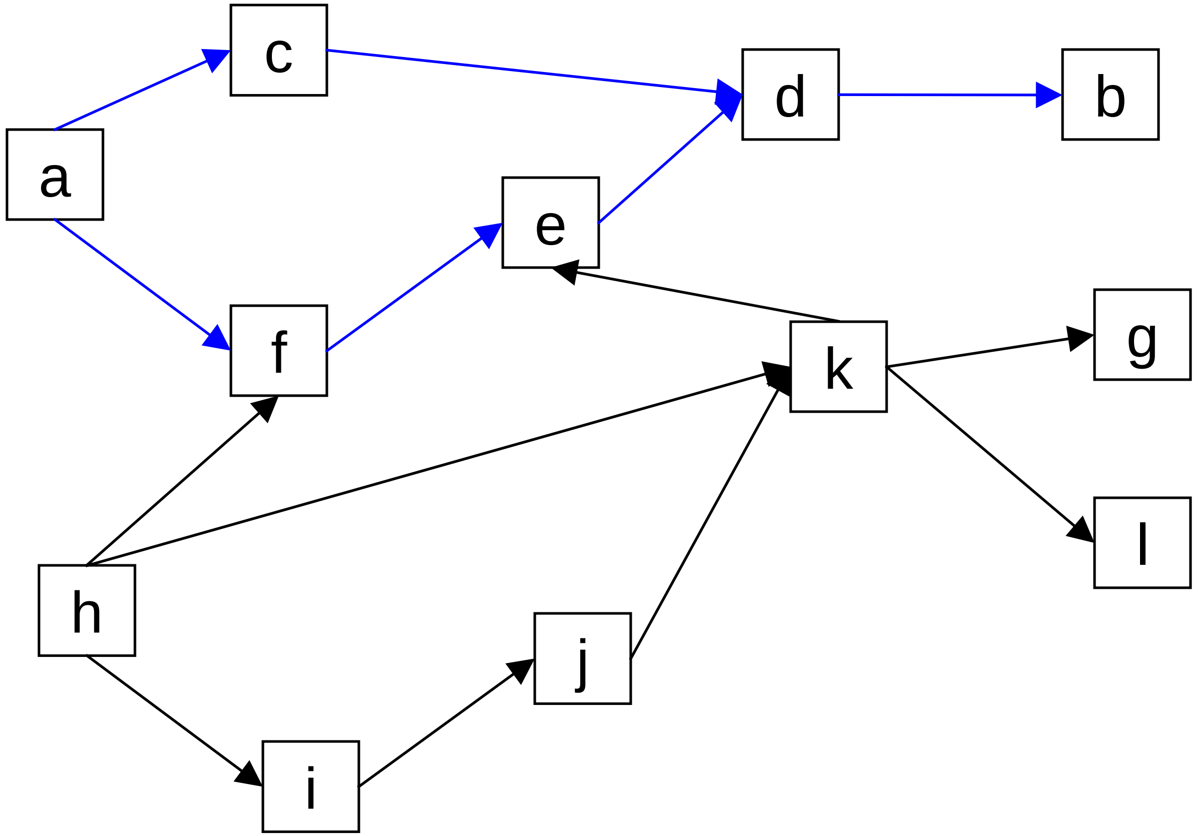
Top-down requires left recursion (EDB subgoals first), as it proves subgoals from left to right. The order is unimportant in bottom-up (conjunction/join is commutative). However, bottom-up computes all paths, although the query concerns only the path from a to b

Datalog vs Prolog: bottom-up vs top-down

Example: recursion, a graph path

p(X, Y) ← e(X, Y). p(X, Y) ← e(X, Z), p(Z, Y). **?- p(a, b).**

Bottom-up computation (Datalog) processes **all paths**, i.e. also paths in the black component of the graph. However, only blue edges matter



Datalog vs Prolog: bottom-up vs top-down

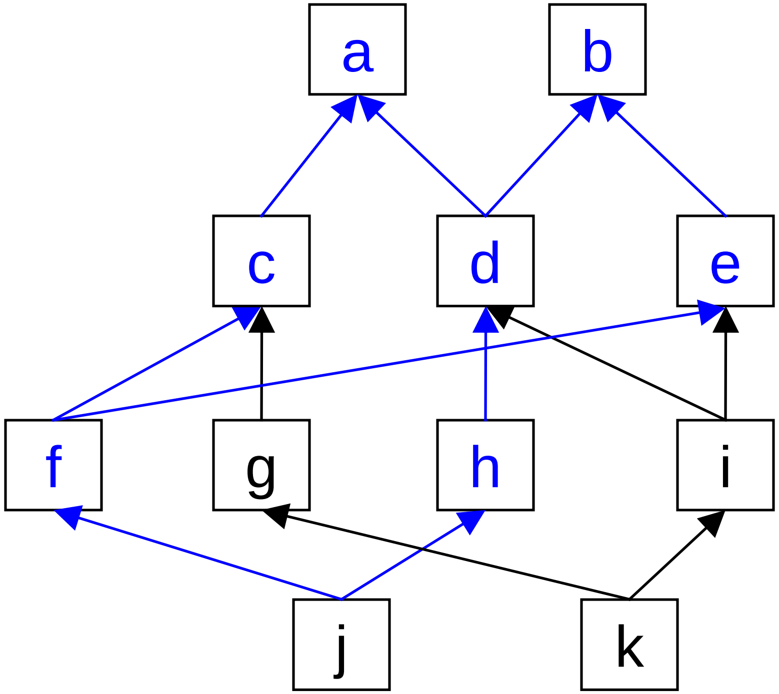
Another example: recursion, ancestors (an acyclic graph)

anc(X, Y) ← par(X, Y). /* Y is parent of X */

anc(X, Y) ← par(X, Z), anc(Z, Y).

?- anc(j, A). /* ancestors of j (j stands for John) */

EDB: par(X, Y) = {[c, a], [c, d], [d, b], [e, b], [f, c], [f, e], [g, c], [h, d], [i, d], [i, e], [j, f], [j, h], [k, g], [k, i]}



Addition of an arbitrary graph below vertices f, g, h, i, j, k does not influence the time complexity of top-down computation of John’s ancestors. But it will influence the time complexity of bottom-up computation (naïve iteration computes the entire relation anc, not only John's ancestors)

Top-down computation: Rule-Goal Tree (RGT)

1. The goal, G_0, is the root node of the tree.
2. The leaves of the root node are all rule nodes applicable to G_0. Heads of these nodes are unified with G_0 so that:
   a) Before the unification, all variables of the rules are made disjoint with variables appearing in the current goal (each rule gets "fresh" numbered variables before processing)
   b) During unification with a head, goal variables are preferred (i.e. they do not disappear)
   c) If a variable in the head is substituted, the substitution is applied to all occurrences of that variable in the rule
   d) Variables not appearing in the head (the fresh variables) are local in the rule (i.e. they do not appear in any other rule)
3. The leaves of a rule node are subgoals of the rule body. Each subgoal is recursively processed in a similar manner as G_0

…

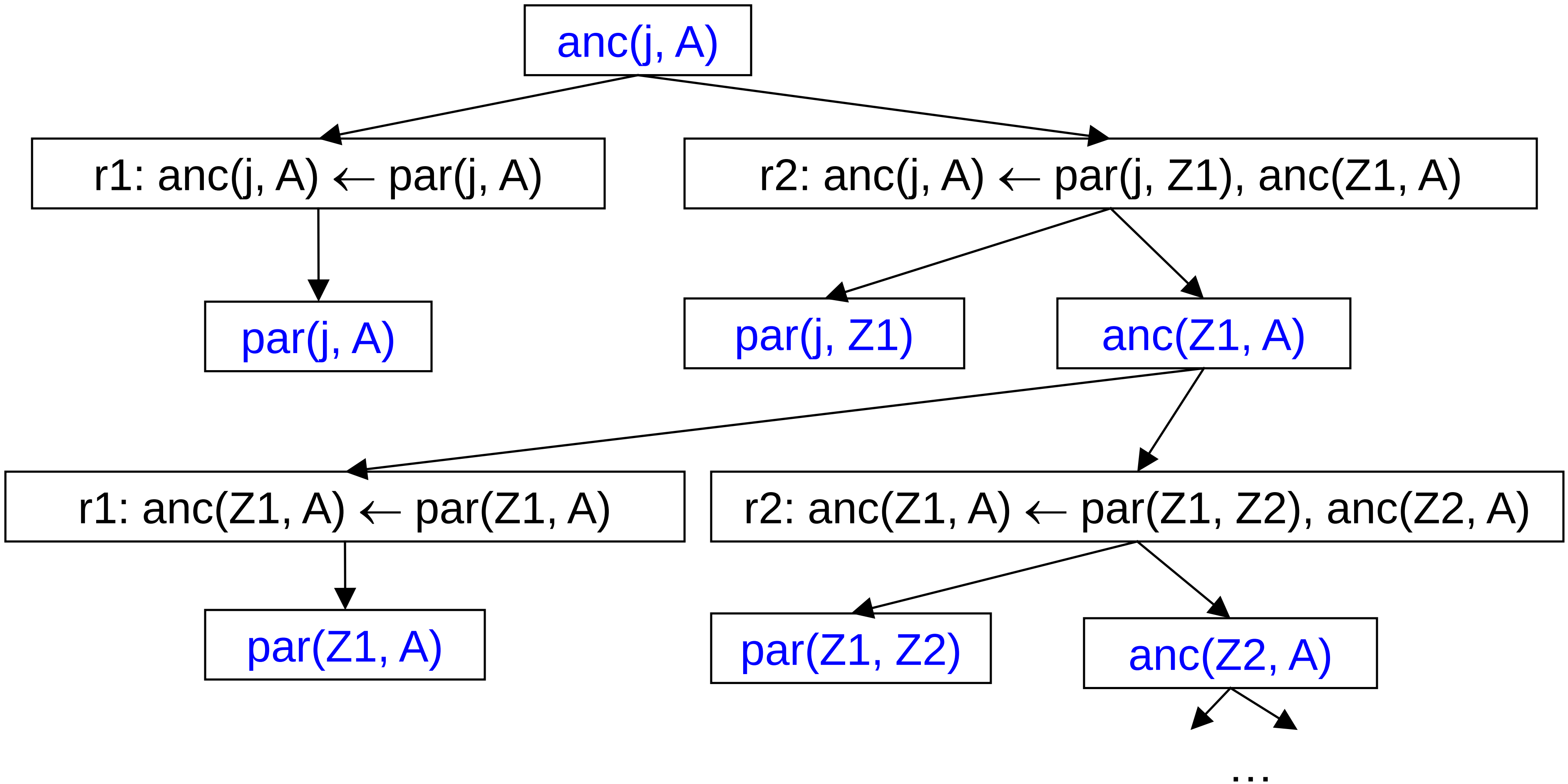
Top-down computation: Rule-Goal Tree (RGT)

Example: recursion, ancestors

r1: anc(X, Y) ← par(X, Y). /* Y is parent of X */

r2: anc(X, Y) ← par(X, Z), anc(Z, Y).

?- anc(j, A). /* ancestors of j (j stands e.g. for John) */



Top-down computation: Rule-Goal Tree (RGT)

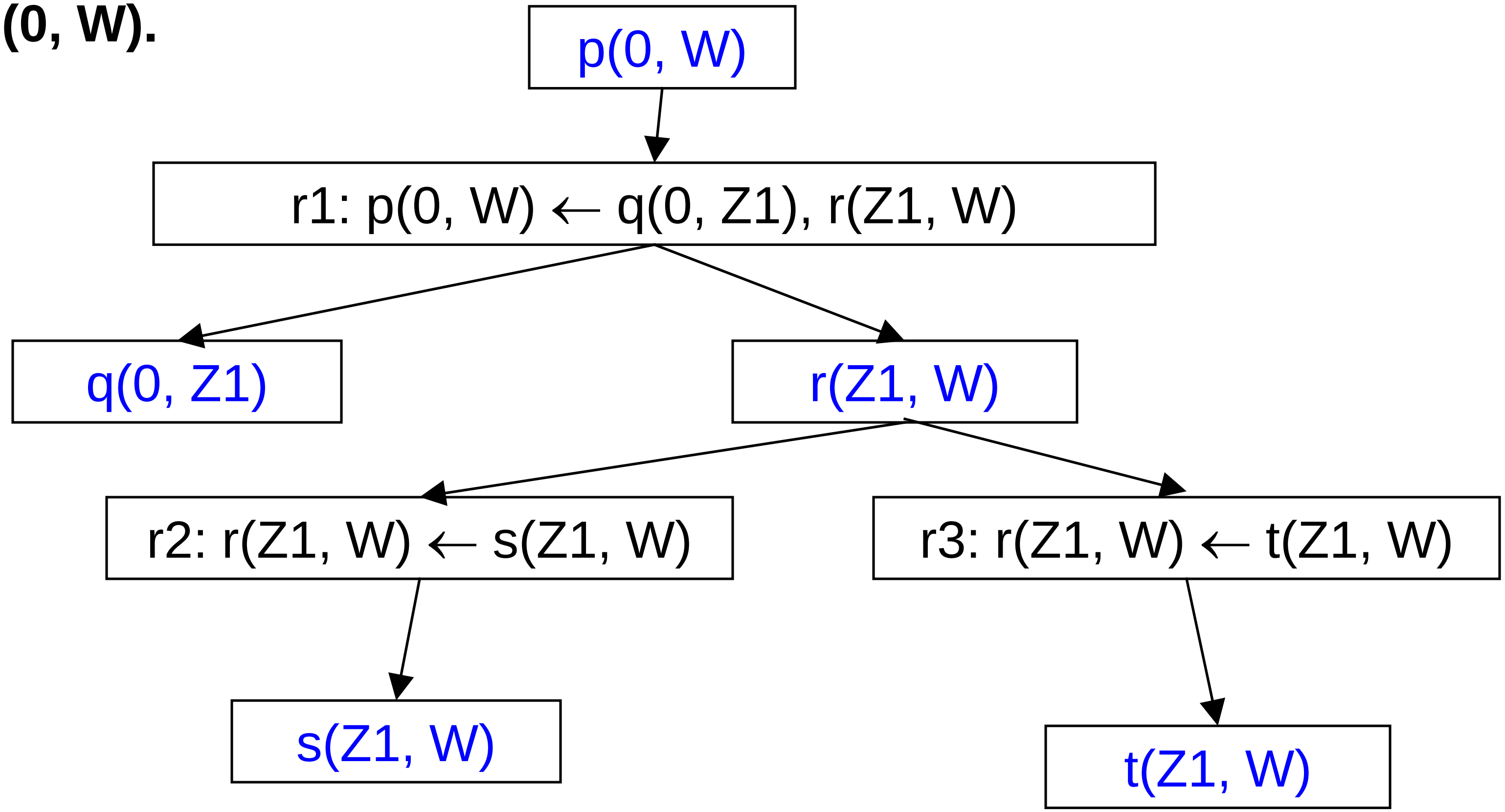
A non-recursive example

r1: p(X, Y) ← q(X, Z), r(Z, Y).

r2: r(U, V) ← s(U, V).

r3: r(U, V) ← t(U, V).

?- p(0, W).



Top-down: propagation of bindings and results in RGT

- Each **goal node** is assigned a **binding relation M (“magic predicate”)**. Bound variables of the goal are attributes of the corresponding magic relation. The contents of the magic relation is a finite set of constant values resulting from the previous computation (“sideway-passed information”)
- In each **rule node, i-th argument** of the rule is assigned a **“supplementary relation” S_i**. Supplementary relations collect sideway-passed information, i.e. possible instantiations for variables of the rule. Initially, this information is determined only by the head of the rule. It is updated as the computation moves from left to right over subgoals of the rule
- **Each goal node as well as each rule node** yield a **resulting relation**. For a goal node, it is the set of tuples which satisfy the goal. For a rule node, it is the set of tuples which satisfy the head of the rule

Top-down: actual computation

- Actual computation takes place in rule nodes
- In the root and inner **goal nodes**, the resulting relation is computed as **union of the results** from the child nodes
- **Leaves** of the tree are **joins** (selections) of the magic relations with EDB relations
- **In rule nodes**, the subgoals are first converted to supplementary relations with variables (using **atov**). The **supplementary relations are joined (or antijoined, in case of negation)**. The result is then converted to head (using **vtoa**) and sent to the parent node

RGT construction and evaluation

- Input: a set of safe rules, EDB and a query (goal) G_0
- Output: a set of all tuples which satisfy G_0
- Method: two mutually recursive functions, **expand_goal** and **expand_rule**
  - expand_goal(M, G, R), where G is a goal, M is the binding relation assigned to G, R is the resulting relation
  - expand_rule(S_0, r, R), where r is a rule, S_0 is the supplementary predicate containing bindings given by the head, R is the resulting relation

For a query ?- G_0, the computation begins with expand_goal(M_0, atov(G, P), R), where M_0 is the binding relation for the variables in G_0, P is the predicate in G_0, R is the result. (For a query with no bindings, M_0 is initialised to a universe, i.e. $S \bowtie M_0 = S$ for an arbitrary relation S.)

Top-down: expand_goal(M, G, R)

```
expand_goal(M, G, R) {
    if (G is a goal with an EDB predicate P)
        R = M ⋈ atov(G, P);
    else {
        /* G is a goal with an IDB predicate */
        R = ∅; /* the result will be accumulated in R */
        for (each rule r whose head H unifies with G) {
            τ = mgu(G, H);
            H' = Π_M (Hτ); /* Hτ converted to arguments of G */
            S_0 = atov(H', M);
            expand_rule(S_0, rτ, R_r);
            R = R ∪ R_r;
    }
}
```

Top-down: expand_rule(S_0, r, R)

```
expand_rule(S_0, r, R) {
    /* let r be of form H ← G_1, …, G_k */
    for (i = 1; i <= k; i++)
    {
        M_i = vtoa(G_i, S_i-1);
        expand_goal(M_i, G_i, R);
        Q_i = atov(G_i, R); /* convert R_i to variables */
        S_i = Π_T (S_i-1 ⋈ Q_i);/* T is the set of variables which
            appear in S_i-1 or Q_i in the rule r */
    }
    R = vtoa(H, S_k);
}
```

Top-down: QRGT algorithm

- Recursive **expand_rule / expand_goal** algorithm is a **depth-first** traversal of the RGT. It can end in an infinite loop
- Modification of the expand_rule / expand_goal algorithm: **QRGT (queue-based rule-goal-tree expansion)**. The idea is that **breadth-first** traversal of RGT does not get lost in an infinite branch
- QRGT computes the fix-point for safe Datalog programs
- For a goal G with magic relation M, QRGT computes a tuple $[t_1, \ldots, t_k]$ when this tuple is computed by the bottom-up computation

Adorned predicates and rules

- **Adornment for a predicate** is a (finite) string of symbols ‘b’ and ‘f’ which stand for ‘bound’ and ‘free’. An adornment states which arguments of the predicate are bound just before the predicate is evaluated (called)
- **Adornment for a rule** (more precisely, for a **position inside a rule**) is a list assigned to “spaces between goals” in the rule, $[V_{b1}, V_{b2}, \ldots V_{bm} \mid V_{f1}, V_{f2}, \ldots, V_{fn}]$. The symbol ‘|’ in the list separates bound and free variables of the rule when the computation just after the top-down computation has reached that point inside the rule. Note that a bound variable inside a rule is restricted to finitely many values. A variable is bound after the goal G_i when it has already been bound in the head of the rule, or when it appears in the rule anywhere before the goal G_i (from left to right, including G_i)

Rule-Goal Graph (RGG)

Rule-Goal Graph consists of goal (predicate) nodes and rule position nodes. RGG is a generalisation of RGT. Unlike RGT, RGG is always finite, but may contain cycles (when a predicate or a rule goal with the same adornment is evaluated more than once during the top-down computation). Edges of RGG connect nodes so that:

- Node with an EDB predicate has no outgoing edges
- Outgoing edges of a node with an adorned IDB predicate p^a end in rule nodes r_0[… | …] such that the head of the rule can be unified with p^a, i.e. they bind the same variables
- Outgoing edges of a node with an adorned rule position r_i[… | …] end in
  a) goal nodes p_i^a, where p_i is the predicate in goal G_i (i.e. the goal following the position r_i) and the sets of variables bound in a and in the adornment of r_i are equal
  b) node r_{i+1} (if r_i is a rule position before the penultimate goal), where variables bound in r_{i+1} are those which have been bound in r_i plus those which appear in the goal G_i

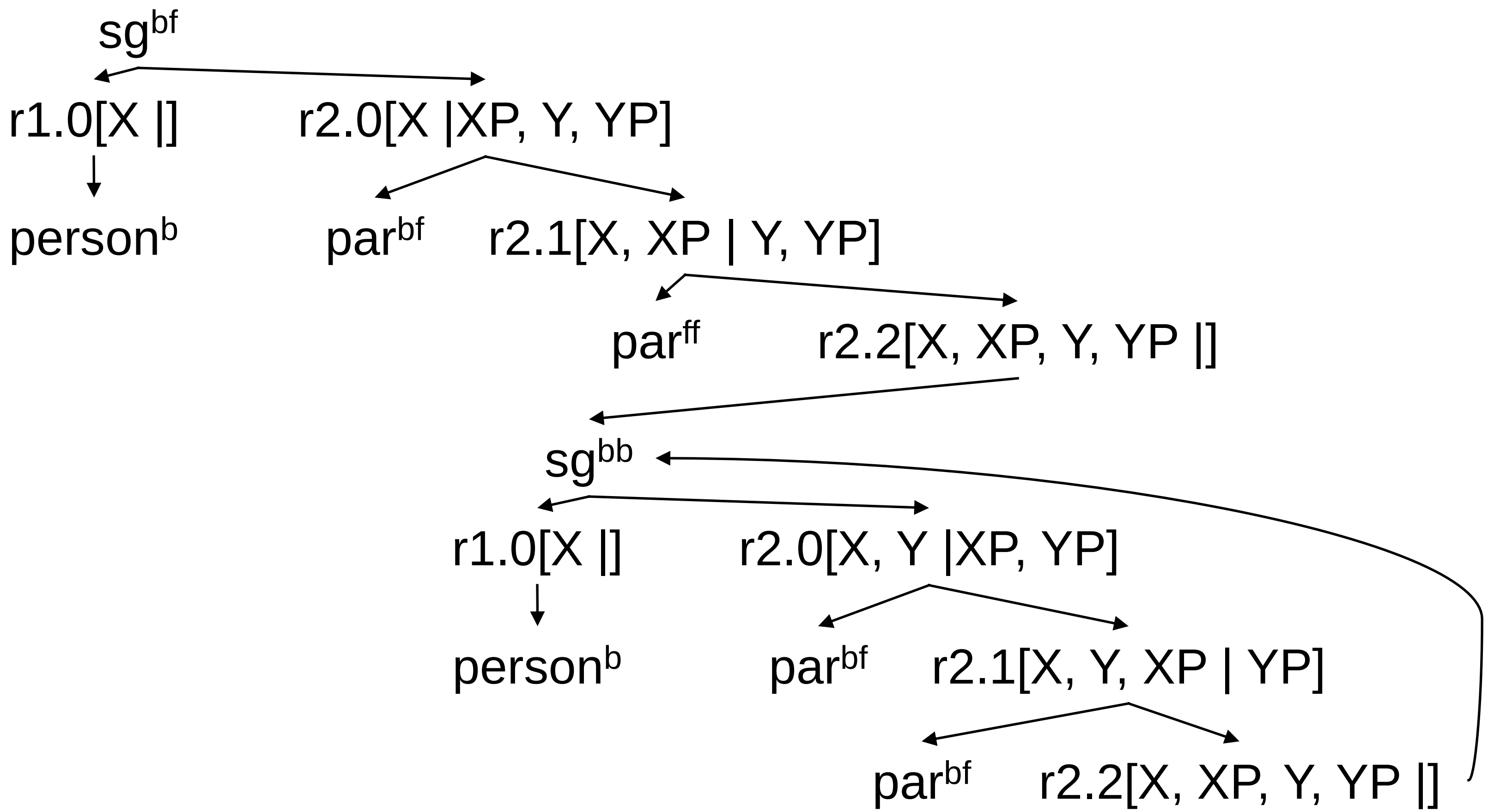
Rule-Goal Graph (RGG)

Example: the same generation (person and parent are EDB predicates)

r1: sg(X, X) ← person(X).

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

?- sg(j, Y).



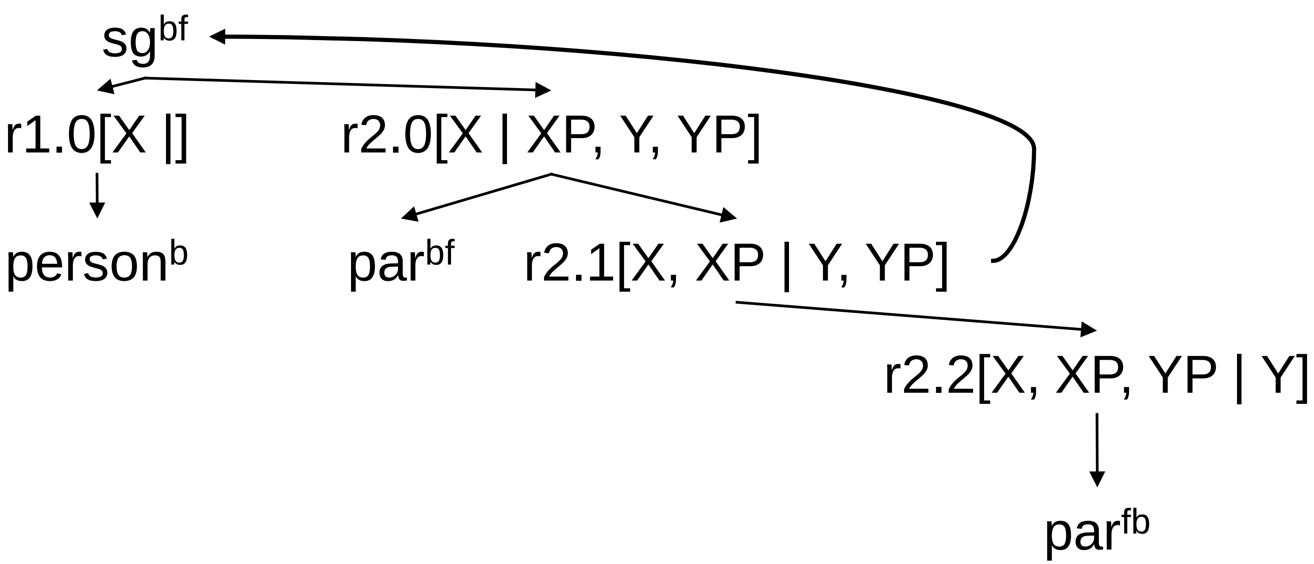
Rule-Goal Graph (RGG)

Example: the same generation with reordered goals:

r1: sg(X, X) ← person(X).

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

?- sg(j, Y).



Rule-Goal Graph (RGG)

Problems arising during the construction of an RGG:

• **Rectification of subgoals in rules**. This is advisable when either a constant or repeated variable appears in an IDB goal. (For example, adornment for p(X, X) is neither p^{ff}, nor p^{bb}, it is actually slightly stronger. Similarly, p(1, 3) is stronger than just p^{bb}.)

• **Making adornments of IDB predicates uniform** by reordering subgoals in rules

• **Making built-in subgoals feasible**. It may be possible to compute built-in subgoals for some adornments, but not all

→ feasibility problem for RGG

Making adornments of IDB predicates uniform

- For each adorned predicate p^α in RGG, create a new predicate p_α
- Copy all rules r with head p into rules r_α with head p_α
- In all goal nodes r_0, …, r_k of the transformed rule r_α, modify its child nodes as follows:

    a) Keep EDB and built-in predicates in the rule r_α as they were in rule r

    b) Change all adorned IDB predicates q^β to q_β

Making adornments of IDB predicates uniform: example

Original program:

r1: sg(X, X) ← person(X).

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

?- sg(j, Y).

Modified program with uniform adornments of IDB predicate sg:

r1_bf: sg_bf(X, X) ← person(X).

r2_bf: sg_bf(X, Y) ← par(X, XP), par(Y, YP), sg_bb(XP, YP).

r1_bb: sg_bb(X, X) ← person(X).

r2_bb: sg_bb(X, Y) ← par(X, XP), par(Y, YP), sg_bb(XP, YP).

?- sg_bf(j, Y).

Rectification

Motivation:

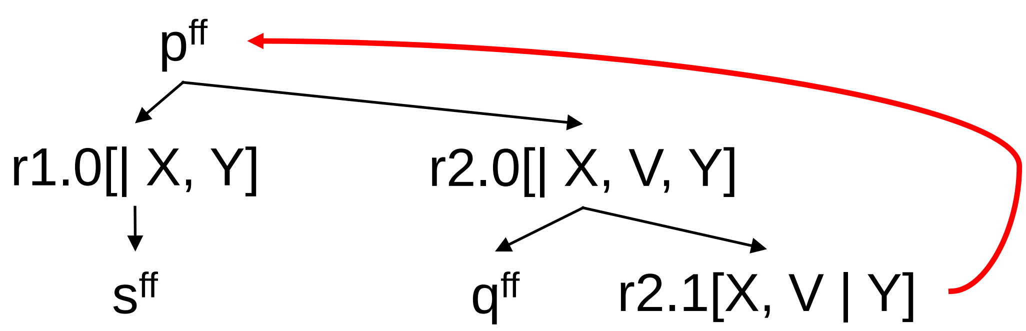
r1: $p(X, Y) \leftarrow s(X, Y)$.

r2: $p(X, Y) \leftarrow q(X, V), p(Y, Y)$.

?- $p(X, Y)$.

Note that the adornment p^{ff} does not fully capture the bindings in the call $p(Y, Y)$ in r2. Although the variable Y is free, the two arguments in p are not arbitrary, they must be equal!

The adornments in the following RGG are not quite exact



Rectification

Rectification is an algorithm which produces a program **without duplicated variables and without constants in IDB subgoals.**

It defines **new predicates** (with fewer arguments):

r1: p(X, Y) ← s(X, Y).

r2: p(X, Y) ← q(X, V), p(Y, Y).

p(X, Y) ← s(X, Y).

p(X, Y) ← q(X, V), p1(Y). /* p1(Y) ← p(Y, Y) */

p1(Y) ← s(Y, Y). /* expansion using r1 */

p1(Y) ← q(Y, V), p1(Y). /* expansion using r2 */

Rectification

while (IDB goal *g* exists with a duplicated variable or a constant in arguments)

{

Let *g* be a goal where a predicate *p* is called. Replace p in the body a new predicate *p'* with no duplicated variables and no constants in arguments

For each rule *r* with head *h*, create a new rule *r'*. To do that, compute τ = *mgu(g, h)*, preferring the variables of *g*. The new rule *r'* is $r\tau$, where all occurences of *p* are replaced with *p'*.

In all the other rules, replace *p* with *p'*.

}

Note that this algorithm always terminates (the number of arguments decreases in each iteration). Moreover, if the original program is safe, so is the rectified program

Ordering of goals in rules

A rule in RGG is evaluated always from left to right. However, we can arbitrarily order the goals in the rule

Sometimes we **must reorder the goals** because of built-in goals

For example, the rule

p(X, Y) ← X < Y, r(X, Y).

equivalently: p(X, Y) ← less(X, Y), r(X, Y).

cannot be evaluated from left to right e.g. for the adornment p^{ff}.

The only adornment allowed in a call to less is $less^{bb}$.

But this ordering of goals is all right:

p(X, Y) ← r(X, Y), less(X, Y).

Ordering of goals in rules

Other reasons for reordering goals:

• **Making use of indexes for EDB**. Some adornments of EDB predicates may profit from using an index attached to a database relation

• **IDB predicates with functional symbols** in their definition may also forbid some adornments. For example, nat^f leads to an infinite expansion, whereas nat^b does not.

Rules for nat(.): nat(0). nat(s(X)) $\leftarrow$ nat(X).

• **Negated IDB goals** require that all their arguments are bound

Ordering of goals in rules

We assume that **bound is easier**

We define a partial ordering of adornments:

$\alpha \leq \beta$ if β has 'b' at least on those positions where α has

Then if $\alpha \leq \beta$ and we can evaluate p^{α}, then we can also evaluate p^{β}

For example, if we can evaluate p^{f}, then we can also evaluate p^{b}

We say that an adornment is **allowed** if the predicate can be evaluated for that adornment

For each predicate, there is a **set for minimal allowed adornments**

Ordering of goals in rules

Assumed that for each adorned rule (i.e. adornment of the head), minimal allowed adornments for each goals are given

How to find an ordering of the goals so that the rule can be evaluated?

Backtracking: enumerate all the orderings and test whether a feasible RGG (only with allowed adornments) can be constructed

Optimisations:

- Assume “bound is easier”
- Maintain two sets of adorned goals: 1.a set *T* of target adorned goals (this initially contains only the query); 2.a set *F* of adorned goals which cannot be realized (this initially contains goals with adornments which must be avoided, i.e. which are not allowed)

Ordering of goals in rules

Example: the same generation

r1: $sg(X, X) \leftarrow person(X)$.

r2: $sg(X, Y) \leftarrow par(X, XP), par(Y, YP), sg(XP, YP)$.

?- sg(john, W).

$T=\{sg^{bf}\}$, $F=\{par^{ff}, person^{f}\}$

There is only one ordering for $r1^{bb}$ which avoids $person^{f}$.

For $r2^{bf}$, this ordering must be tested: par_1^{bf}, sg^{fb}, par_2^{fb}. Therefore, sg^{fb} is added into the set T. It remains to show that sg^{fb} can be realised.

There is only one ordering for $r1^{bb}$ which avoids $person^{f}$.

For $r2^{fb}$, the following ordering is realisable: par_2^{bf}, sg^{bf}, par_1^{fb}.

We end up with $T=\{sg^{bf}, sg^{fb}\}$. All goals in T are realisable

Ordering of goals in rules

Example: the same generation, ?- sg(john, W). **F={par^{ff}, person^{f}}**

r1: sg(X, X) $\leftarrow$ person(X).

r2^{bf}: sg(X, Y) $\leftarrow$ par(X, XP), sg(XP, YP), par(Y, YP).

r2^{fb}: sg(X, Y) $\leftarrow$ par(Y, YP), sg(XP, YP), par(X, XP).

Feasible RGG: