Compilation and Optimization

CS240B Notes

Notes based on Section 9.6 of *Advanced Database Systems*—Morgan Kaufmann, 1997

C. Zaniolo, April 2002

Rule-Goal Graph

- The graph has as nodes rules with *adorned* predicate names.
- The adornment of the predicate is the subscript that denotes bound/free argument.
- The graph depicts all possible top-down, left-to-right executions.

Example

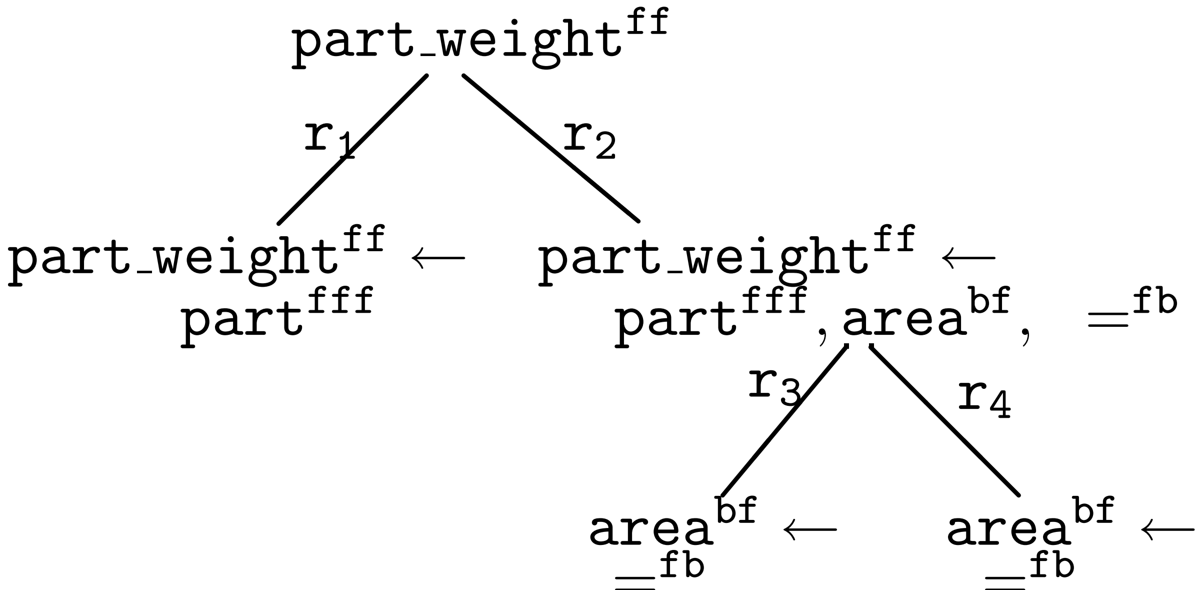
$$r_1 : \mathtt{part\_weight}(\mathtt{No}, \mathtt{Kilos}) \leftarrow \quad \mathtt{part}(\mathtt{No}, \mathtt{\_}, \mathtt{actualkg}(\mathtt{Kilos})).$$

$$r_2 : \mathtt{part\_weight}(\mathtt{No}, \mathtt{Kilos}) \leftarrow \quad \mathtt{part}(\mathtt{No}, \mathtt{Shape}, \mathtt{unitkg}(\mathtt{K})),$$

$$\mathtt{area}(\mathtt{Shape}, \mathtt{Area}), \mathtt{Kilos} = \mathtt{K} * \mathtt{Are}$$

$$r_3 : \mathtt{area}(\mathtt{circle}(\mathtt{Dmtr}), \mathtt{A}) \leftarrow \quad \mathtt{A} = \mathtt{Dmtr} * \mathtt{Dmtr} * 3.14/4.$$

$$r_4 : \mathtt{area}(\mathtt{rectangle}(\mathtt{Base}, \mathtt{Height}), \mathtt{A}) \leftarrow \quad \mathtt{A} = \mathtt{Base} * \mathtt{Height}.$$



$\mathtt{part\_weight}^{ff}$
$\mathtt{r}_1$
$\mathtt{r}_2$
$\mathtt{part\_weight}^{ff} \leftarrow \mathtt{part}^{fff}$
$\mathtt{part\_weight}^{ff} \leftarrow \mathtt{part}^{fff}, \mathtt{area}^{bf}, =^{fb}$
$\mathtt{r}_3$
$\mathtt{r}_4$
$\mathtt{area}^{bf} \leftarrow =^{fb}$
$\mathtt{area}^{bf} \leftarrow =^{fb}$

Construction of the rule-goal graph $rgg(P)$ for a nonrecursive program P

1. Initial step: *The query goal is adorned according to the constants and deferred constants (i.e., the variables preceded by \$), and becomes the root of $rgg(P)$.*
2. Bindings passing from goals to rule heads: *If the calling goal g unifies with the head of the rule r, with mgu γ, then we draw an edge (labeled with the name of the rule, i.e., r) from the adorned calling goal to the adorned head, where the adornments for $h(r)$ are computed as follows: (i) all arguments bound in g are marked bound in $h(r)\gamma$; (ii) all variables in such arguments are also marked bound; and (iii) the arguments in $h(r)\gamma$ that contain only constants or variables marked bound in (ii) are adorned b, while the others are adorned f.*

Rule-Goal graph for nonrecursive P–cont

1. *Left-to-right passing of bindings to goals:*
   A variable X is bound after the n^{th} goal in a rule, if X is among the bound head variables (as for the last step), or if X appears in one of the goals of the rule preceding the n^{th} goal.
   The $(n+1)^{th}$ goal of the rule is adorned on the basis of the variables that are bound after the n^{th} goal.

Unification of Goal and Head: example

Goal $?g$, with: $g = p(f(X_1), Y_1, Z_1, a)$

Rule: $r : p(X_2, g(X_2, Y_2), Y_2, W_2) \leftarrow \ldots$.

A most general unifier for g and $h(r)$ is:

$\gamma = \{X_2/f(X_1), Y_1/g(f(X_1), Y_2), Z_1/Y_2, W_2/a\}$

Yielding: $g\gamma = h(r)\gamma = h(r) = p(f(X_1), g(f(X_1), Y_2), Y_2, a)$

If the adorned goal is p^{bffb}: variables in the first argument of the head (i.e., X_1) are bound. The resulting adorned head is p^{bffb}, and there is an edge from p^{bffb} to $p^{bffb} \leftarrow$.

If the adorned goal is p^{fbfb}: all the variables in the second argument of the head (i.e., X_1, Y_2) are bound. Then the remaining arguments of the head are bound as well. Draw an edge from the adorned goal p^{fbfb} to the adorned head $p^{bbbb} \leftarrow$.

Safety

Safe a-priori:

1. Base predicates are safe for every adornment. E.g., $\texttt{part}^{\texttt{fff}}$ is safe.
2. $\theta^{\texttt{bb}}$ is safe for θ denoting any comparison operator, such as $\leq$ or $>$.
3. Special cases: $=^{\texttt{bf}}$ or $=^{\texttt{fb}}$ where the free argument consists of only one variable: the arithmetic expression in the bound argument can be computed and the resulting value can be assigned to the free variable.(A more sophisticated compiler could solve more equations and accept other patterns as safe.)

Inductive Definition for Safety

Let P be a program with rule-goal graph $rgg(P)$, where $rgg(P)$ is a tree (DAGs can be reduced to trees):
Then P is safe if the following two conditions hold:

1. Every leaf node of $rgg(P)$ is safe a priori, and
2. Every variable in every rule in $rgg(P)$ is bound after the last goal.

Computation of a Safe Nonrecursive Program with rule graph $rgg(P)$

1. First phase: the bound values of a goal's arguments are passed to its defining rules, i.e., its children in the rule-goal graph.
2. Second phase: the goal receives the values of the f-adorned arguments from its children.

Only the second computation takes place for goals without bound arguments.

The heads of the rules are computed once all the goals in the body are computed. We have a strict strati£cation where predicates are computed according to the postorder traversal of the rule-goal graph.

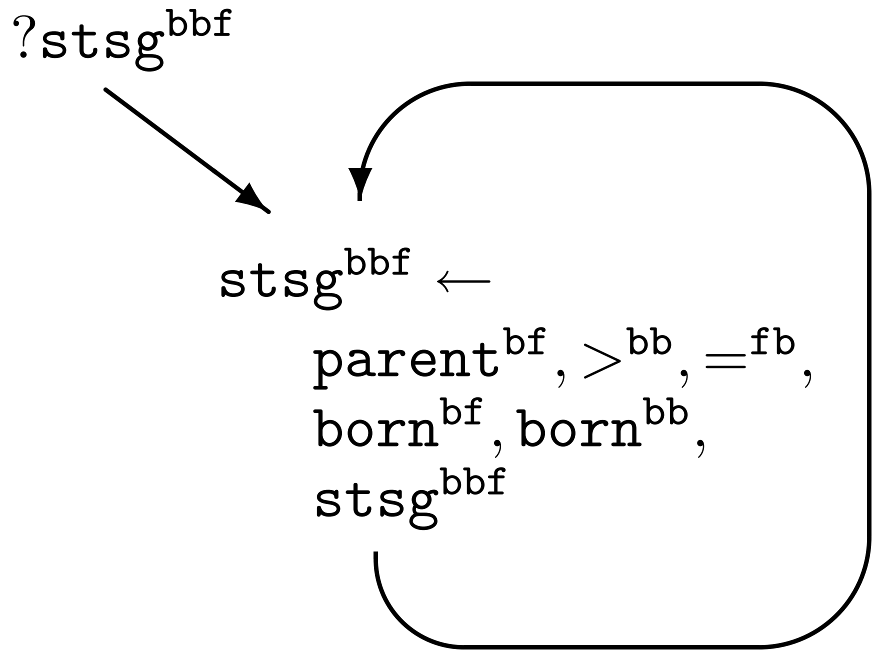
Recursive Predicates

A choice of recursive methods must be performed along with the binding passing analysis.
CASE1: no bound argument. The recursive predicate and all the predicates that are mutually recursive with it are be computed in a single differential fixpoint.
The construction of the rule graph for a recursive rule is the same as for a non-recursive one.

1. The head of the rule is assumed to have no bound argument, and
2. Safety analysis is performed by treating the recursive goals (i.e., p and predicates mutually recursive with it) as safe a priori—in fact, they are bound to the values computed in the previous step.

Binding passing analysis for recursive predicates

The Same-Generation query:


$?\mathtt{stsg}^{\mathtt{bbf}}$
$\mathtt{stsg}^{\mathtt{bbf}} \leftarrow$
$\mathtt{parent}^{\mathtt{bf}}, >^{\mathtt{bb}}, =^{\mathtt{fb}},$
$\mathtt{born}^{\mathtt{bf}}, \mathtt{born}^{\mathtt{bb}},$
$\mathtt{stsg}^{\mathtt{bbf}}$

Only *chain goals* are used in the top-down propagation.

q^γ *is a chain goal if:*

1. **SIP independence of recursive goals:** q is not a recursive goal (i.e., not the same predicate as that in the head of r, nor a predicate mutually recursive with q; however, recursive predicates of lower strata can be used as chain goals).
2. **Selectivity:** q^γ has some argument bound
3. **Safety:** q^γ is a safe goal.

The basic idea behind the notion of chain goals is that the binding in the head will have to reduce the search space. Any goal that is called with all its adornment free will not be beneficial in that respect.
Also, there is no sideway information passing (SIP) between two recursive goals; bindings come only from the head through nonrecursive goals.

Safety–Cont

- If q is not a recursive predicate, then safety is determined as previously described.
- If q is a recursive goal, then it belongs to a lower stratum; therefore, safety can be determined independently using the techniques described here for recursive predicates.
- Since we have a finite number of strata the process terminates.

Algorithm for Binding Passing Analysis

1. Initially $\mathbf{A} = \{q^\gamma\}$, with q^γ the initial goal, where q is a recursive predicate and γ is not a totally free adornment.
2. For each $h \in \mathbf{A}$, pass the binding to the heads of rules defining q.
3. For each recursive rule, determine the adornments of its recursive goals (i.e., of q or predicates mutually recursive with q).
4. If the last step generated adornments not currently in $\mathbf{A}$, add them to $\mathbf{A}$ and resume from step 2. Otherwise halt.

Binding Passing Property

The calling goal g is said to have the

1. *binding passing property* when $\mathbf{A}$ does not contain any recursive predicate with totally free adornment.
2. *Unique binding passing property*: if binding passing property holds and $\mathbf{A}$ contains one pattern for each recursive predicate.

Selecting a Method for Recursion

Using the rewriting for the magic sets method, which can then be used as the basis for left/right-linear rules. E.g.:

$$?\mathtt{anc}(\mathtt{tom}, \mathtt{Desc}).$$

$$\mathtt{anc}(\mathtt{Old}, \mathtt{Young}) \leftarrow \mathtt{parent}(\mathtt{Old}, \mathtt{Young}).$$

$$\mathtt{anc}(\mathtt{Old}, \mathtt{Young}) \leftarrow \mathtt{anc}(\mathtt{Old}, \mathtt{Mid}), \mathtt{parent}(\mathtt{Mid}, \mathtt{Young}).$$

$$\mathtt{m.anc}(\mathtt{tom}).$$

$$\mathtt{m.anc}(\mathtt{Old}) \leftarrow \mathtt{m.anc}(\mathtt{Old}).$$

The magic relation `anc` now contains only `tom`. We can substitute this value directly into the rules.The recursive magic rule is trivial and can be eliminated: trivial first phase.

Trivial Second Phase for right-linear rules

$$
\begin{array}{ll}
?\mathtt{anc}(\mathtt{tom}, \mathtt{Desc}). & \\
\mathtt{anc}(\mathtt{Old}, \mathtt{Young}) \leftarrow & \mathtt{father}(\mathtt{Old}, \mathtt{Young}). \\
\mathtt{anc}(\mathtt{Old}, \mathtt{Young}) \leftarrow & \mathtt{parent}(\mathtt{Old}, \mathtt{Mid}), \mathtt{anc}(\mathtt{Mid}, \mathtt{Young}).
\end{array}
$$

$$
\begin{array}{ll}
\mathtt{m.anc}(\mathtt{tom}). & \\
\mathtt{m.anc}(\mathtt{Mid}) \leftarrow & \mathtt{m.anc}(\mathtt{Old}), \mathtt{parent}(\mathtt{Old}, \mathtt{Mid}). \\
\mathtt{anc}'(\mathtt{Old}, \mathtt{Young}) \leftarrow & \mathtt{m.anc}(\mathtt{Old}), \mathtt{father}(\mathtt{Old}, \mathtt{Young}). \\
\mathtt{anc}'(\mathtt{Old}, \mathtt{Young}) \leftarrow & \mathtt{parent}(\mathtt{Old}, \mathtt{Mid}), \mathtt{anc}'(\mathtt{Mid}, \mathtt{Young}), \\
& \mathtt{m.anc}'(\mathtt{Old}). \\
?\mathtt{anc}'(\mathtt{tom}, \mathtt{Young}). &
\end{array}
$$

The recursive rule only copies the value of $\mathtt{Young}$ generated by the exit rule, from the tail to the head. This value of $\mathtt{Young}$ is returned as an answer if, after a few iterations, $\mathtt{Old} = \mathtt{tom}$. This is always true since this rule re-visits the magic-set computation.

Trivial Second Phase–Cont

Thus the recursive rule can be dropped along with the condition in the first argument of the query goal, yielding: After dropping the recursive rule can be dropped along with the condition in the first argument of the query goal, we obtain:

$$\begin{array}{ll}
\mathtt{m.anc}(\mathtt{tom}). & \\
\mathtt{m.anc}(\mathtt{Mid}) \leftarrow & \mathtt{m.anc}(\mathtt{Old}), \mathtt{parent}(\mathtt{Old}, \mathtt{Mid}). \\
\mathtt{anc}'(\mathtt{Old}, \mathtt{Young}) \leftarrow & \mathtt{m.anc}(\mathtt{Old}), \mathtt{father}(\mathtt{Old}, \mathtt{Young}). \\
?\mathtt{anc}'(\_, \mathtt{Young}). &
\end{array}$$

Generalizations

- *Unique binding property*. Relaxing this assumption does not require major modifications or extensions
- *No Sideway Information Passing (SIP)* between recursive goals: only goals from lower strata can be used as chain goals
  This assumption can be removed yielding the *Generalized Magic Set method*.
  The programs produced by this extension tend to be complex and inefficient to execute.
- In the CORAL system, not all the variables are required to be instantiated after a goal executes.

Optimization

In relational databases there are two kinds of optimizations

1. *Greedy optimization*: whenever a technique is applicable apply it. E.g., always push selection and projection into relational expressions. Computationally this is not very expensive
2. *Cost Based Optimization*: evaluate alternatives and predict the cost. Then choose the least-expected-cost solution. This is done for choosing a join order. Basically exponential in the number of joins being evaluated.

Deductive Database prototypes follow mostly the first approach.

Selecting a method for recursion

1. The binding passing property is tested, and if satisfied
2. The applicability of the following methods are considered in the order shown:
   1. left- right-linear rules

   [1.5 Counting Method]

   2. magic or supplementary magic sets

   [2.5 Generalized magic sets methods]

Most systems do not bother with [1.5] or [2.5].

Optimization—cont.

1. Ideally, the cost/benefits of different recursive methods should be quantified and compared. But quantification is expensive and prediction is unreliable.
   In practice, therefore, very coarse criteria are used instead—e.g., chain goals in the SIP.
2. Even for nonrecursive rules, full cost-based optimization is problematic (many goals deeply stacked). Heuristics approaches are used instead. E.g., Glue/Nail! uses the following Heuristic: *Do First goals with more bound argument; and between two goals with the same number of bound arguments, select those which have fewer unbound arguments*.
3. Following the order of goals specified by the user— in $\mathcal{LDL}$++ and Prolog.

Existential Variables: Optimization

Existential Variables.

$$\mathtt{p}(\mathtt{X}) \leftarrow \quad \mathtt{q1}(\mathtt{X}, \mathtt{Y}), \mathtt{q2}(\mathtt{Y}, \mathtt{Z}), \neg\mathtt{q3}(\mathtt{W})$$

If $\mathtt{q2}(\mathtt{Y}, \mathtt{Z})$ succeeds or fails for certain value of $\mathtt{Y}$, there is no need to find all the other values of $\mathtt{Y}$.
Same for $\mathtt{W}$. $\mathtt{Y}$ and $\mathtt{W}$ are existential variables.
A tuple a tuple-oriented model of computation:

(i) Get-first tuple in relation (joining with the previous tuples, if any)
(ii) Get-next of same, and repeat this step till no more such tuples

For $\mathtt{q1}(\mathtt{X}, \mathtt{Y})$ both steps must me performed.
For $\mathtt{q3}(\mathtt{W})$ and $\mathtt{q2}(\$\mathtt{Y}, \mathtt{Z})$ only step (i) is executed.

Three-Way Join

$$\leftarrow \mathtt{p1}(\mathtt{X},\mathtt{Y}), \mathtt{p2}(\mathtt{Y},\mathtt{Z}), \mathtt{p3}(\mathtt{Y},\mathtt{W})$$

The basic nested-loop join:

```
Loop 1: for each tuple in p1 do
        Loop 2: for each tuple in p2 (joining with p1) do
                Loop 3: for each tuple in p3
                        (joining with p1 and p2) do
                        return the computed tuple
                end Loop 3
        end Loop 2
end Loop 1.
```

But if $\mathtt{p3}(\mathtt{Y},\mathtt{W})$ *fails on first (i.e., step (i) fails) there is no point in going back to* Loop 2, *since only a new value of* $\mathtt{Y}$ *can make it succeed!*

Strength and Weakness of Deductive DB Technology

A rather powerful and sophisticated technology to support rules and queries on Databases. But problems, such as optimization, require some work. But the main problems seem to be limitations in terms of expressive power.

- Does not support negation and aggregates in recursion
- Active rules are not supported and updates are only supported as imperative extensions.