Knowledge-Based Systems and Deductive Databases

Wolf-Tilo Balke
Christoph Lofi
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de

8.0 Datalog to RelAlg

• Datalog can be converted to Relational Algebra and vice versa
  – This allows to merge Datalog-style reasoning with relational databases
    • e.g. Datalog on RDBs, Recursive SQL, etc.
  – The elementary production rule (and thus the fixpoint iteration) has been implemented with relational algebra in the last lecture

8.0 Datalog to RelAlg

• In addition to bottom-up approaches (like fixpoint iteration), there are also top-down evaluation schemes for Datalog
  – Idea: Start with query and try to construct a proof tree down to the facts
  – Simple Bottom Up approach: Construct all possible search trees by their depth
    • Search tree: Parameterized proof tree
      – Search tree can be transformed to a proof tree by providing a valid substitution

8.0 Datalog to RelAlg

– Search tree are constructed by backwards-chaining of rules
– Problem: When to stop?
  • A naive solution: Compute the theoretical maximal chain length and use as limit
– Outlook for today: Optimization techniques
  • Evaluation optimization
  • Query rewriting

Exercise 2

• Fixpoint iteration
  – path(X,Y) :- edge(X,Y)
  – path(X,Y) :- edge(X,Y), path(Z,Y)

New facts added by proof tree length!
Exercise 3.2

Stratification
- \( q(1,2) \)
- \( q(2,3) \)
- \( q(1,3) \)
- \( r(X,Y) : \neg s(X,Y) \)
- \( p(X,Y) : q(X,Y), \neg r(X,Y) \)
- \( p(X,Y) : q(X,Y), \neg s(X,Y) \)
- \( p(X,Y) : p(X,Y), p(X,Y) \)

Exercise 3.5

Translate Datalog\textsuperscript{neg} to Datalog. Idea:
- Use closed world assumption
- Introduce new predicates
- \( ns(X,Y) = \text{false} \) for \( X=1, Y=3, \text{true otherwise} \)
- \( nr(X,Y) : \neg ns(X,Y) \)

8.1 Query Optimization

The computation algorithms introduced in the previous weeks were all far from optimal
- Usually, a lot of unnecessary deductions were performed
- Wasted work
- Termination problems, etc...
- Thus, this week we will focus on optimization methods

Search Technique:
- Bottom-Up
  - Start with extensional database and use forward-chaining of rules to generate new facts
  - Result is subset of all generated facts
  - Set oriented-approach → Very well-suited for databases
- Top-Down
  - Start with queries and either construct a proof tree or a refutation proof by backward-chaining of rules
  - Result is generated tuple-by-tuple → More suited for complex languages, but less desirable for use within a database

Furthermore, there are two possible (non-exclusive) formalisms for query optimization
- Logical: A Datalog program is treated as logical rules
  - The predicates in the rules are connected to the query predicate
  - Some of the variables may already be bound by the query
- Algebraic: The rules in a Datalog program can be translated into algebraic expressions
  - Thus, the IDB corresponds to a system of algebraic equations
  - Transformations like in normal database query optimization may apply
8.1 Query Optimization

- Optimizations can address different objectives
  - Program Rewriting:
    - Given a specific evaluation algorithm, the Datalog program \( \mathcal{P} \) is rewritten into a semantically equivalent program \( \mathcal{P'} \)
    - However, the new program \( \mathcal{P'} \) can be executed much faster than \( \mathcal{P} \) using the same evaluation method
  - Evaluation Optimization:
    - Improve the process of evaluation itself, i.e., program stays as it is but the evaluation algorithm is improved
    - Can be combined with program rewriting for even increased effect

- When optimizing, two approaches are possible
  - Syntactic: just focus on the syntax of rules
    - Easier and thus more popular than semantics
    - E.g., restrict variables based on the goal structure or use special evaluation if all rules are linear, etc.
  - Semantic: utilize external knowledge during evaluation
    - E.g., integrity constraints
    - External constraints: “Lufthansa flights arrive at Terminal 1”;
      Query: “Where does the flight LH1243 arrive?”

- Not all combinations are feasible or sensible
  - We will focus on following combinations

<table>
<thead>
<tr>
<th>BOTTOM-UP</th>
<th>TOP-DOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation</td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td>Naïve (Jacobi, Gauss-Seidel)</td>
<td>Naïve Top-Down with search trees</td>
</tr>
<tr>
<td>Semi-naïve (Delta Iteration)</td>
<td>Query-Subquery</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search technique</td>
<td>bottom-up, top-down</td>
</tr>
<tr>
<td>Formalism</td>
<td>logic, relational algebra</td>
</tr>
<tr>
<td>Objective</td>
<td>rewriting, pure evaluation</td>
</tr>
<tr>
<td>Traversal order</td>
<td>depth-first, breadth-first</td>
</tr>
<tr>
<td>Approach</td>
<td>syntactic, semantic</td>
</tr>
<tr>
<td>Structure</td>
<td>rule structure, goal structure</td>
</tr>
</tbody>
</table>

- Optimization techniques may be combined
  - Thus, mixed execution of rewriting and evaluation techniques based on logical and algebraic optimization is possible
    - Start with logic program \( \mathcal{L} \)

- Optimizations can focus on different traversal-orders
  - Depth-First
    - Order of the literals in the body of a rule may affect performance
      - E.g., consider top-down evaluation with search trees for
        \[ P(X,Y): P(X,X), Q(Y,Z) \] vs. \[ P(X,Y): Q(Z,Y), P(X,X) \]
      - In more general cases (e.g., Prolog), may even affect decidability
      - It may be possible to quickly produce the first answer
  - Breadth-First
    - Whole right-hand-side of rules is evaluated at the same time
    - Search trees grow more balanced
    - Due to the restrictions in Datalog, this becomes a set-oriented operation and is thus very suitable for DB's

- Summary of optimization classification with their (not necessarily exclusive) alternatives
8.1 Query Optimization

Transformation into Relational Algebra

Logical query evaluation methods

Algebraic query evaluation methods

Query result

8.2 Evaluation Methods

• Evaluation methods actually compute the result of an (optimized or un-optimized) program \( \mathcal{P} \)

<table>
<thead>
<tr>
<th>Evaluation Method</th>
<th>BOTTOM-UP</th>
<th>TOP-DOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>Jacobi, Gauss-Seidel</td>
<td>Henschen-Napvi</td>
</tr>
<tr>
<td>Semi-naïve (Delta Iteration)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

— Better evaluation methods skip unnecessary evaluation steps and/or terminate earlier

8.2 Bottom-Up Evaluation

• Datalog programs can easily be evaluated in a bottom-up fashion, but this should also be efficient
  — The naïve algorithm derives **everything that is possible** from the facts
  — But naively answering queries wastes valuable work...
  — For dealing with recursion we have to evaluate fixpoints
    • For stratified Datalog\( ^{\text{top}} \) programs we apply the fixpoint algorithms to every stratum

8.2 Bottom-Up Evaluation

• Bottom-up evaluation techniques are usually based on the **fixpoint iteration**
  • Remember: Fixpoint iteration itself is a **general concept** within all fields of mathematics
    — Start with an **empty initial solution** \( X_0 \)
    — Compute a new \( X_{n+1} \) from a given \( X_n \) by using a production rule
      • \( X_{n+1} := T(X_n) \)
      — As soon as \( X_{n+1} = X_n \), the algorithm stops
      • Fixpoint reached

8.2 Bottom-Up Evaluation

• Up to now we have stated the **elementary production** rule declaratively
  — \( T_p : I \rightarrow \{ B \in B \mid \text{there exists a ground instance} \}
    \( B : A_p, A_p \ldots, A_p \text{ of a program clause such that} \)
    \( \{ A_p, A_p \ldots, A_p \} \subseteq I \)
• However, we need an operative implementation
  — The set \( I_{n+1} \) is computed from \( I_n \), as follows:
    • Enumerate all ground instances \( G \)
      — Each ground instance is given by some substitution (out of a finite set)
    • Iterate over the ground instances, i.e. try all different substitutions
      — For each \( B : A_p, A_p \ldots, A_p \in G \), if \( \{ A_p, A_p \ldots, A_p \} \subseteq I_n \), add \( B \) to \( I_{n+1} \)

8.2 Bottom-Up Evaluation

• **Full Enumeration**: Consecutively generate and test all instances by enumeration
  • Loop over all rules
    — Apply each possible substitution on each rule

**Constant symbols**: \( \{1,2,3\} \)

**Rules**:
\[ p(X,Y) \land r(X,Y) \land p(X,Y) \land r(X,Y) \land p(X,Y) \]

**Enumeration of instances**:

**Rule 1**:
\[ p(1,1) \land r(1,1), \ p(1,2) \land r(1,2), \ p(1,3) \land r(1,3), \ p(2,1) \land r(2,1), \ p(2,2) \land r(2,2), \ p(2,3) \land r(2,3), \ p(3,1) \land r(3,1), \ p(3,2) \land r(3,2), \ p(3,3) \land r(3,3) \]

**Rule 2**:
\[ p(1,1) \land r(1,1), \ p(1,2) \land r(1,2), \ p(1,3) \land r(1,3), \ p(2,1) \land r(2,1), \ p(2,2) \land r(2,2), \ p(2,3) \land r(2,3), \ p(3,1) \land r(3,1), \ p(3,2) \land r(3,2), \ p(3,3) \land r(3,3) \]
8.2 Bottom-Up Evaluation

b) Restricted enumeration

- Loop over all rules
  - For each rule, generate all instances possible when trying to unify the rules right hand side with the facts in \( I \)
  - Only instances which will trigger a rule in the current iteration will be generated

Constant symbols: \( \{1,2,3\} \)
Rules: \[ p(X,Y) : e(X,Y), p(X,Y) : e(X,Z), p(Z,Y) \]
\[ i(X,Y): (e(1,2), e(2,2), e(2,3)) \]
Enumeration of instances:
Rule 1:
\[ p(1,2) : e(1,2), p(2,3) : e(2,3) \]
Rule 2: Nothing, \( p(X,Y) \) can not be unified with any fact in \( I \)

8.2 Jacobi Iteration

- Both fixpoint iterations introduced previously in the lecture are Jacobi iterations
  - i.e. fixpoint iteration and iterated fixpoint iteration
  - i.e. \( I_{n+1} := T \in I_n \)
    - “Apply production rule to all elements in \( I_n \) and write results to \( I_{n+1} \). Repeat”

8.2 Gauss-Seidel Iteration

- Idea:
  - The convergence speed of the Jacobi iteration can be improved by also respecting intermediate results of current iteration
  - This leads to the class of Gauss-Seidel-iterations
  - Historically, an improvement of the Jacoby equation solver algorithm
  - Devised by Carl Friedrich Gauss and Philipp Ludwig von Seidel
  - Base property:
    - If new information is produced by current iteration, it should also be possible to use it the moment it is created (and not starting next iteration)

- A Gauss-Seidel fixpoint iteration is obtained by modifying the elementary production
  - \( T \in I \rightarrow \{ B \in B \mid \text{there exists a ground instance which has not been tested before in this iteration B : } A_1, A_2, ..., A_n \text{ of a program clause such that } \{A_1, A_2, ..., A_n \} \subseteq \{I \cup \text{new_B's}\} \}
  - \text{new_B's} refers to all heads of the ground instances of rules considered in the current iteration which had their body literals in \( I \)
    - Some of these are already in \( I \), but others are new and would usually only be available starting next iteration \( \rightarrow \) improved convergence speed

8.2 Gauss-Seidel Iteration

- The most naïve fixpoint algorithm class are the so-called Jacobi-iterations
  - Developed by Carl Gustav Jacob Jacobi for solving linear equitation systems \( Ax = b \), early 19th century
  - Characteristics:
    - Each intermediate result \( X_{n+1} \) is wholly computed by utilizing all data in \( X_n \)
    - No reuse between both results
    - Thus, the memory complexity for a given iteration step is roughly \( |X_n| |X_n| \)

8.2 Jacobi Iteration

- Please note
  - Within each iteration, all already deduced facts of previous iteration are deduced again
  - Yes, they were… We just used the union notation for convenience
  - \( I_{0} := I \cup \{e(1,2), e(1,3)\} \) was actually not reflecting this correctly
  - \( I_{2} := \{e(1,2), e(1,3), p(1,2), p(1,3)\} \) matches algorithm better…
  - Furthermore, both sets \( I_{n+1} \) and \( I_n \) involved in the iteration are treated strictly separately
  - Elementary production checks which rules are true within \( I_n \) and puts result into \( I_{n+1} \)
Example program $\mathcal{P}$

- edge(1, 2).
- edge(1, 3).
- edge(2, 4).
- edge(3, 4).
- edge(4, 5).
- path(X, Y) := edge(X, Y).
- path(X, Y) := edge(X, Z), path(Z, Y).

1. $I_0 = \emptyset$
2. $I_1 = \{(1, 2), (1, 3), (2, 4), (3, 4), (4, 5)\}$
3. $I_2 = \{(1, 5)\}$

Please note:
- The effectiveness of Gauss-Seidel iteration for increasing convergence speed varies highly with respect to the chosen order of instance enumeration
- e.g. “Instance $K$ tested - generates the new fact $B_1$ from $I_i$”, “Instance $I$ tested - generates the new fact $B_1$ from $I_i \cup B_1$”
- Good luck - improvement vs. Jacobi
- v.s. “Instance $I$ tested - does not fire because it needs fact $B_1$”, “Instance $K$ tested - generates the new fact $B_1$ from $I_i$”
- Bad luck - no improvement

8.2 Gauss-Seidel Iteration

Each single iteration which can be saved improves performance dramatically as each iteration recomputes all known facts!

For both Gauss-Seidel and Jacobi, a lot of wasted work is performed
- Everything is recomputed times and again
- But it can be shown that the elementary production rule is strictly monotonic
- Thus, each result is a subset of the next result
- i.e. $I_i \subseteq I_{i+1}$

This leads to the semi-naive evaluation for linear Datalog

For linear Datalog systems and deductive databases based systems and deductive databases, the semi-naive evaluation is a preferred method for evaluating queries because it avoids re-computing known facts, but makes sure that at least one of the facts in the body of a rule is new if a new fact is computed!

- Idea: avoid re-computing known facts, but make sure that at least one of the facts in the body of a rule is new, if a new fact is computed!
- Really new facts, always involve new facts of the last iteration step, otherwise they could already have been computed before...

It is important to efficiently calculate $\Delta I_{i+1} := T_\mathcal{P}(I_i \cup \Delta I_i) \setminus I_i$
- Especially the $T_\mathcal{P}$ operator is often inefficient, because it simply applies all rules in the EDB
- More efficient is the use of auxiliary functions
- Define an auxiliary function of $T_\mathcal{P}$: aux$_\mathcal{P}$: $2^{|\mathcal{P}|} \times 2^{\mathcal{P}} \rightarrow 2^{|\mathcal{P}|}$ such that $T_\mathcal{P}(I_i \cup \Delta I_i \setminus I_i)$
- Auxiliary functions can be chosen intelligently by just taking recursive parts of rules into account.
- A classic method of deriving auxiliary functions is symbolic differentiation

8.2 Semi-Naïve Evaluation
### 8.2 Semi-Naïve Evaluation

- **The symbolic differentiation operator** $dF$ can be used on the respective relational algebra expressions $E$ for Datalog programs.

  - $dF(E) := \Delta R$, if $E$ is an IDB or EDB relation $R$.

  - $dF(\sigma_{\theta}(E)) = \sigma_{\theta}(dF(E))$ and $dF(\pi_{\theta}(E)) = \pi_{\theta}(dF(E))$.

  - $dF(E_1 \cup E_2) = dF(E_1) \cup dF(E_2)$.

  - This is interesting, especially since delta sets of extensional predicates are empty.

  - Not affected by selections, projections, and unions.

- **Consider the program**
  - ancestor(X, Y) :- parent(X, Y).
  - ancestor(X, Y) :- parent(Z, X), ancestor(Z, Y).
  - The respective expression in relational algebra for ancestor is $\pi_{\#1, \#2}(\sigma_{#2=\#1}(\pi_{#1, \#2}(parent \bowtie ancestor)))$.

- **Symbolic differentiation**
  - $dF(\pi_{\#1, \#2}(\pi_{#1, \#2}(parent \bowtie ancestor))) = dF(\pi_{\#1, \#2}(\sigma_{#2=\#1}(\pi_{#1, \#2}(parent \bowtie ancestor))))$.

- **Having found a suitable auxiliary function the delta iteration** works as follows.
  - Initialization:
    - $I_1 := \emptyset$.
    - $\Delta_1 := \sigma_{\#2=\#1}(\emptyset)$.

  - Iteration until $\Delta_{i+1} = \emptyset$:
    - $I_{i+1} := I_i \cup \Delta_i$.
    - $\Delta_{i+1} := aux_\#1(I_i \cup \Delta_i) \setminus I_i$.

- **Again, for stratified Datalog** programs the iteration has to be applied to every stratum.

- **Let’s consider our ancestor program again**

#### George Sonja Peter Karen

- parent(Thomas, John).
- parent(Mary, John).
- parent(George, Thomas).
- parent(Sonja, Thomas).
- parent(Peter, Mary).
- parent(Karen, Mary).

- ancestor(X, Y) :- parent(X, Y).
- ancestor(X, Y) :- parent(Z, X), ancestor(Z, Y).

- Aux ancestor(ancestor, ancestor) := $\pi_{\#1, \#2}(\sigma_{#2=\#1}(\pi_{#1, \#2}(parent \bowtie ancestor)))$.

#### Thomas Mary

- ancestor := $\emptyset$.
- $\Delta_{\text{ancestor}} := T_{\#2}(\emptyset)$.
- $= ((T, J), (M, J), (G, T), (S, T), (P, M), (K, M))$.

- $\Delta_{\text{ancestor}} := \pi_{\#1, \#2}(parent \bowtie ancestor) \setminus ancestor$.
- $= ((G, I), (S, J), (P, J), (K, J))$. 

#### John

- $\Delta_{\text{ancestor}} := aux_{\#2}(\text{ancestor \bowtie ancestor}) \setminus$ ancestor.
- $= ((G, I), (S, J), (P, J), (K, J))$. 

8.2 Semi-Naïve Evaluation

\[ \text{ancestor}_2 := \text{ancestor}_1 \cup \Delta \text{ancestor}_2 \]
\[ = \{(T, J), (M, J), (G, T), (S, T), (P, M), (K, M), (G, J), (S, J), (P, J), (K, J)} \]
\[ \Delta \text{ancestor}_3 := \text{aux}_{\text{ancestor}_1} (\text{ancestor}_2) \setminus \text{ancestor}_2 \]
\[ := \pi_{#1, #2}(\text{parent} \bowtie_{#2=#1} \text{ancestor}_2) \setminus \text{ancestor}_2 \]
\[ = \emptyset \]

Thus, the least fixpoint is \( \text{ancestor}_2 \cup \text{parent} \).

8.2 Push Selection

• Transforming a Datalog program into relational algebra also offers other optimizations
- Typical relational algebra equivalences can be used for heuristically constructing better query plans
  - Usually an operator tree is built and transformed
  - Example: push selection
    - If a query involves a join or Cartesian product, pushing all selections down to the input relations avoids large intermediate results
  - But now we have a new operator in our query plan: the least fixpoint iteration (denoted as LFP).

8.2 Push Selection

- Consider an example
  - edge(1, 2).
  - edge(4, 2).
  - edge(2, 3).
  - edge(3, 5).
  - edge(5, 6).
  - path(X, Y) :- edge(X, Y).
  - path(X, Y) :- edge(X, Z), path(Z, Y).
  - Relational algebra: edge \( \cup \pi_{#1, #2}(\text{edge} \bowtie_{#2=#1} \text{path}) \)

8.2 Push Selection

- To answer the query we now only have to consider the facts and rules having the correct second argument
  - edge(2, 3).
  - path(2, 3).
  - path(1, 3).
  - path(4, 3).
  - Result: \{2, 1, 4\}

8.2 Push Selection

- Now let's try a different query \(?\text{path}(3, Y)\) \( \pi_{#1} \sigma_{#2=3}(LFP (\text{edge} \cup \pi_{#1, #2}(\text{edge} \bowtie_{#2=#1} \text{path})))) \)
  - From which nodes there is a path to node 3?
  - The above query binds the first argument of path
  - \text{path}(X, Y) :- edge(X, Y).
  - \text{path}(X, Y) :- edge(X, Z), path(Z, Y).
  - Thus the selection could be pushed down to the edge and path relations

8.2 Push Selection

- Now consider the query \(?\text{path}(X, 3)\) \( \pi_{#1} \sigma_{#1=3}(LFP (\text{edge} \cup \pi_{#1, #2}(\text{edge} \bowtie_{#2=#1} \text{path})))) \)
  - To which nodes there is a path from node 3?
  - The above query binds the second argument of path
  - \text{path}(X, Y) :- edge(X, Y).
  - \text{path}(X, Y) :- edge(X, Z), path(Z, Y).
8.2 Push Selection

• To answer the query we now only have to consider the facts and rules having the correct second argument

\[
\begin{align*}
\text{edge}(3,5) & \quad \text{fact} \\
\text{path}(3,5) & \quad \text{r}_1 \\
\emptyset & \quad \text{r}_2 \\
\end{align*}
\]

Result: \{5\}

Obviously this is wrong

8.3. Logical Rewriting

• In the following, we deal with rewriting methods

<table>
<thead>
<tr>
<th>Rewriting Method</th>
<th>Logic</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magic Sets</td>
<td></td>
<td>Variable reduction</td>
</tr>
<tr>
<td>Counting</td>
<td></td>
<td>Constant reduction</td>
</tr>
<tr>
<td>Static Filtering</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Basic Idea:
  - Transform program \( P \) to a semantically equivalent program \( P' \) which can be evaluated faster using the same evaluation technique
  - e.g. same result, but faster when applying Jacobi iteration

8.3. Magic Sets

• Magic Sets
  - Magic sets are a rewriting method exploiting the syntactic form of the query
  - The base idea is to capture some of the binding patterns of top-down evaluation approaches into rewriting
    • If there is a subgoal with a bound argument, solving this subgoal may lead to new instantiations of other arguments in the original rule
    • Only potentially useful deductions should be performed

8.3. Logical Rewriting

• Clever rewriting could work like this:

\[
P: \\
\text{ancestor}(X, Y) \leftarrow \text{parent}(X, Y). \\
\text{ancestor}(X, Y) \leftarrow \text{ancestor}(Wolfi, Z), \text{parent}(Z, Y). \\
\text{ancestor}(Wolfi, Y)?
\]

• This simple transformation will skip the deduction of many (or in this case all) useless facts
• Actually, this transformation was straightforward and simple, but there are also unintuitive but effective translations…
  • Magic sets!
8.3. Magic Sets

- Who are the ancestors of Wolfi?

Raphael  Maria  Possibly unimportant
George  Sonja  Peter  Karen
Wolf
Mary
They
Sarah
Paul
Tiffy

- A typical top-down search tree for the goal ancestor(Wolfi, X) looks like this
  - Possible substitutions already restricted

Q \equiv \text{ancestor}(Wolfi, X)
\text{anc.}(Wolfi, X) :\ - \text{anc.}(Wolfi, Z), \text{par.}(Z, X).
\text{anc.}(Wolfi, Z)
\text{par.}(Z, X)
\text{par.}(Wolfi, Z)

- How can such a restriction be incorporated into rewriting methods?

8.3. Magic Sets

- For rewriting, propagating binding is more difficult than using top-down approaches
- Magic Set strategy is based on augmenting rules with additional constraints (collected in the magic predicate)
  - This is facilitated by “adorning” predicates
  - Sideways information passing (SIP) is used to propagate binding information

8.3. Logical Rewriting

- Arguments of predicates can be distinguished
  - Distinguished arguments have their range restricted by either constants within the same predicate or variables which are already restricted themselves
  - i.e.: The argument is distinguished if
    - it is a constant
    - OR it is bound by an adornment
    - OR it appears in an EDB fact that has a distinguished argument

- Predicates occurrences are distinguished if all its arguments are distinguished
  - In case of EDB facts, either all or none of the arguments are distinguished
  - Predicate occurrences are then adorned (i.e. annotated) to express which arguments are distinguished
  - Adornments are added to the predicate, e.g. \( p^a(X, Y) \) vs. \( p^b(X, Y) \)
8.3. Magic Sets

- For each argument, there are two possible adornments
  - b for bound, i.e. distinguished variables
  - f for free, i.e. non-distinguished variables
- Thus, for a predicate with n arguments, there are $2^n$ possible adorned occurrences
  - e.g. $p_b(X, Y), p_b(X, Y), p_b(X, Y), p_f(X, Y)$
  - Those adorned occurrences are treated as if they were different predicates, each being defined by its own set of rules

Example output of magic set algorithm

P:
ancestor(Wolfi, X)
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).

P:
magic(Wolfi).
magic(Z) :- magic(Y), parent(Z, Y).
ancestor(Wolfi, X) :- magic(X), parent(Y, X).
ancestor(Wolfi, X) :- magic(X), ancestor(Wolfi, X), parent(Z, Y).

8.3. Magic Sets

- The idea of the magic set method is that the magic set contains all possibly interesting constant values
  - The magic set is recursively computed by the magic rules
- Each adorned predicate occurrence has its own defining rules
  - In those rules, the attributes are restricted according to the adornment pattern to the magic set

8.3. Magic Sets

- Incorporate goal query
  - ancestor(X, Wolfi)?
  - ancestor(X, Y) :- parent(X, Y).
  - ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).
  - q(X) :- ancestor(X, Wolfi).
  - ancestor(X, Y) :- parent(X, Y).
  - ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).

- Adorn predicate occurrences
  - q(X) :- ancestor(Wolfi, X).
  - ancestor(Wolfi, X) :- parent(X, Y).
  - ancestor(Wolfi, X) :- ancestor(Wolfi, X), parent(Z, Y).

8.3. Magic Sets

- Now, following problems remain
  - How is the magic set computed?
  - How are the rules for adorned predicate occurrences actually defined?
  - Before solving these problems, we have to find out which adorned occurrences are needed
  - Thus, the reachable adorned system has to be found
    - i.e. incorporate the query as rule and replace all predicate by its respective adornments

8.3. Magic Sets

- For defining the magic set, we create magic rules
  - For each adorned predicate occurrence in a rule of an intensional DB predicate, a magic rule corresponding to the right hand side of that rule is created
    - Predicate occurrences are replaced by magic predicate, bound arguments are used in rule head, free ones are dropped
    - Magic predicates in the head are annotated with its origin (rule & predicate), those on the right hand side just with the predicate
      - q(X) :- ancestor(Wolfi, X, Wolfi).
      - ancestor(Wolfi, X) :- ancestor(Wolfi, X), parent(Z, Y).
      - ancestor(Wolfi, X) :- ancestor(Wolfi, X, Z), parent(Z, Y).
      - magic_r0_ancestor(Wolfi).
      - magic_r2_ancestor(Z) :- magic_ancestor(Z), parent(Z, Y).
Thus, we obtain multiple magic predicates for a single adorned predicate occurrence
- Depending on the creating rule
  - e.g. magic_r0_ancestor^b, magic_r2_ancestor^b both using magic_ancestor^b
- Now we need complementary rules connecting the magic predicates
  - Adorned magic predicate follows from special rule magic predicate with same adornment
    - magic_ancestor^b(X): magic_r0_ancestor^b(X), magic_ancestor^b(X): magic_r2_ancestor^b(X).

As all magic sets are defined, the original rules of the reachable adorned system have to be restricted to respect the sets
- Every rule using an adoned IDB predicate in its body is augmented with an additional literal containing the respective magic set
  - e.g.
    - ancestor^b(X, Y): ancestor^b(X, Z), parent(Z, Y).
    - ancestor^b(X, Y): ancestor^b(X, Z), parent(Z, Y).
    - magic_ancestor^b(X), ancestor^b(X, Z), parent(Z, Y).

Finally, we have a complete definition of magic predicates with different adornments
- In our case, we have only the fb-adornment
  - magic_r0_ancestor^b(Wolf).
  - magic_r2_ancestor^b(Z): magic_ancestor^b(Z), parent(Z, Y).
  - magic_ancestor^b(X): magic_r0_ancestor^b(X), magic_r2_ancestor^b(X).
  - magic_ancestor^b(X): magic_r0_ancestor^b(X), magic_r2_ancestor^b(X).
- The magic magic_ancestor^b set thus contains all possibly useful constants which should considered when evaluating an ancestor subgoal with the second argument bound for the current program
  - Like, e.g. our query...

Finally, the following program is created
ancestor(X, Y) : parent(X, Y).
ancestor(X, Y) : ancestor(X, Z), parent(Z, Y).
ancestor(X, Y) : ancestor^b(X, Y), ancestor^b(X, Z), parent(Z, Y).
ancestor(X, Y) : ancestor^b(X, Y), ancestor^b(X, Z), parent(Z, Y).
ancestor^b(X, Y) : ancestor^b(X, Y), ancestor^b(X, Z), parent(Z, Y).
ancestor^b(X, Y) : ancestor^b(X, Y), ancestor^b(X, Z), parent(Z, Y).
q'(X) : ancestor^b(X, Wolf).

In this example, following further optimizations are possible
- In this case, it is not necessary to separate the two occurrences of magic_r0_ancestor^b and magic_r2_ancestor^b
  - No dependencies between both
  - We can unify and rename them
  - We have only one adornment pattern (fb) and can thus drop it
- This final program can be evaluated using any evaluation technique with increased performance

Magic Sets in short form
- Query is part of the program
- Determine reachable adorned system
  - i.e. observe which terms are distinguished and propagate the resulting adornments
  - Reachable adorned system contains separated adorned predicate occurrences
- Determine the magic set for each adorned predicate occurrence
  - Use magic rules and magic predicates
- Restricts rules using adorned predicates to using only the constant in the respective magic set
• Uncertain Reasoning!