Logic Programming and Deductive Databases

Chapter 14: SLDMagic: Magic Sets and SLD-Resolution

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Summer 2021

`http://www.informatik.uni-halle.de/~brass/lp21/`

Objectives

After completing this chapter, you should be able to:

- compare magic sets with SLD resolution.
- name some problems of the magic set method and sketch possible solutions.

Contents

1. Introduction
2. Tail Recursion
3. Further Problems of Magic Sets
4. The SLDMagic Method

Bottom-Up vs. Top-Down Evaluation (1)

- In this course, two main methods for evaluating logic programs were studied:
  - SLD-Resolution, which works “top-down”.

    The query is seen at the “top”, the given facts at the “bottom”.
  - Bottom-Up evaluation with magic sets.

    We first studied bottom-up evaluation without magic sets, but it is not goal-directed, which in many cases makes it obviously inferior to SLD-resolution (but it always terminates!). The “magic set” transformation solves this problem.
- Of course, we want to know: Which method is faster?

  Not only by running some benchmarks for concrete implementations, but the goal must be also to gain theoretical insights on the efficiency of the two methods.

Bottom-Up vs. Top-Down Evaluation (6)

- It will turn out that
  - every magic fact that is derivable from the transformed program corresponds to a selected literal in the SLD tree,
  - every derivable non-magic fact is proven als a “lemma” in the SLD-tree.
- So there is a strong connection between the two methods.
- However, the implicit representation of lemmas in the SLD-tree can make SLD-resolution significantly faster in case of tail-recursive programs.
- This and other problems can be solved with the SLDMagic technique (developed by the author) presented in the last part of the chapter.

Repetition: Magic Set Transformation (1)

Example (Grandparents of Julia):

- Logic Program (IDB-Predicates and Query):

$$\begin{array}{lcl}
\texttt{parent}(\texttt{X},\texttt{Y}) & \leftarrow & \texttt{mother}(\texttt{X},\texttt{Y}). \\
\texttt{parent}(\texttt{X},\texttt{Y}) & \leftarrow & \texttt{father}(\texttt{X},\texttt{Y}). \\
\texttt{grandparent}(\texttt{X},\texttt{Z}) & \leftarrow & \texttt{parent}(\texttt{X},\texttt{Y}) \wedge \\
 & & \texttt{parent}(\texttt{Y},\texttt{Z}). \\
\texttt{answer}(\texttt{X}) & \leftarrow & \texttt{grandparent}(\texttt{julia},\texttt{X}).
\end{array}$$

- EDB-Predicates (stored in the database):
  - father
  - mother

Repetition: Magic Set Transformation (2)

Example Output, First Part:

- Rules are restricted by an additional body literal so that they can fire only if there is a matching (sub-)query:

  E.g. m_parent_bf(X): There is a query of the form parent(X, _) with given X.

```
parent(X, Y)        ←  m_parent_bf(X) ∧
                       mother(X, Y).
parent(X, Y)        ←  m_parent_bf(X) ∧
                       father(X, Y).
grandparent(X, Z)   ←  m_grandparent_bf(X) ∧
                       parent(X, Y) ∧
                       parent(Y, Z).
answer(X)           ←  true ∧
                       grandparent(julia, X).
```

Repetition: Magic Set Transformation (3)

Example Output, Second Part:

```
m_grandparent_bf(julia)  ←  true.
m_parent_bf(X)           ←  m_grandparent_bf(X).
m_parent_bf(Y)           ←  m_grandparent_bf(X) ∧
                            parent(X, Y).
```

- Of course, the original query `grandparent(julia, X)` must be represented as a magic fact `m_grandparent_bf(julia)` in the rewritten program.
- In addition, magic facts corresponding to the occurring subqueries must be derivable.

  Example: To compute the grandparents of X, one must first compute the parents of X. This is encoded in the second rule above.

Contents

1 Introduction

2 Tail Recursion

3 Further Problems of Magic Sets

4 The SLDMagic Method

Problem: Tail Recursion (1)

Example:

- Computation of all nodes reachable from a given node (standard "transitive closure" example):

$$
\begin{aligned}
\texttt{path}(\texttt{X},\texttt{Y}) &\leftarrow \texttt{edge}(\texttt{X},\texttt{Y}).\\
\texttt{path}(\texttt{X},\texttt{Z}) &\leftarrow \texttt{edge}(\texttt{X},\texttt{Y}) \land \texttt{path}(\texttt{Y},\texttt{Z}).
\end{aligned}
$$

$$?\ \texttt{path}(\texttt{0},\ \texttt{X}).$$

- EDB-Relation (directed graph, in this case one long path):

$$\texttt{edge} := \Big\{(i-1, i) \;\Big|\; 1 \le i \le n\Big\}.$$

$$\begin{tikzcd} \textcircled{0} \arrow[r] & \textcircled{1} \arrow[r] & \cdots \arrow[r] & \textcircled{n} \end{tikzcd}$$

Problem: Tail Recursion (2)

Complexity of Magic Sets:

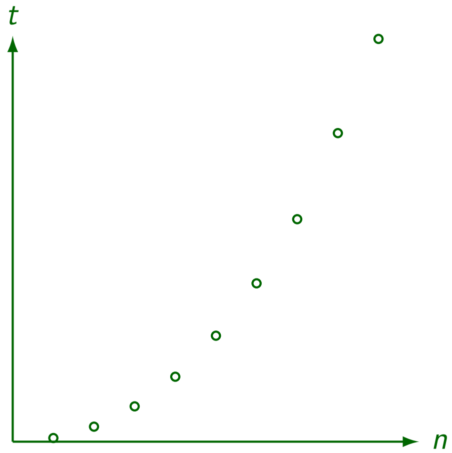
- The query $\texttt{path}(0, \texttt{X})$ calls the subquery $\texttt{path}(1, \texttt{X})$, and so on.
- In the end, all facts $\texttt{path}(i, j)$ are computed.
- Derivable facts:

$$\begin{array}{lll} \texttt{m\_path\_bf}(i) & \text{für} & 0 \le i \le n \\ \texttt{path}(i, j) & \text{für} & 0 \le i < j \le n \end{array}$$

- These are $(n + 2)(n + 1)/2$ facts, thus the runtime is at least quadratic (probably more).
- This example should run in linear time!

Problem: Tail Recursion (3)

Runtime (CORAL):



Problem: Tail Recursion (4)

SLD-Resolution (Prolog):

- The SLD-tree is shown on the next slide.
- Number of nodes in the SLD-tree: $4n + 3$.
- Each node consists of maximally two literals.
- Complexity of each access to `edge`: $O\big(\log(n)\big)$.
- Total complexity: $O\Big(n * \log(n)\Big)$.
- If one could access `edge_bf` in time $O(1)$ (e.g., hash table), the total runtime would really be linear.

Problem: Tail Recursion (5)

$$\begin{tikzcd}
 & \mathtt{path}(0,\mathtt{X}) \arrow[dl, dash] \arrow[dr, dash] & & \\
\mathtt{edge}(0,\mathtt{X}) \arrow[d, dash] & & \mathtt{edge}(0,\mathtt{Y}) \wedge \mathtt{path}(\mathtt{Y},\mathtt{X}) \arrow[d, dash] & \\
\mathtt{X} = 1 & & \mathtt{path}(1,\mathtt{X}) & \\
 & & \vdots & \\
 & & \mathtt{path}(n,\mathtt{X}) \arrow[dl, dash] \arrow[dr, dash] & \\
 & \mathtt{edge}(n,\mathtt{X}) & & \mathtt{edge}(n,\mathtt{Y}) \wedge \mathtt{path}(\mathtt{Y},\mathtt{X})
\end{tikzcd}$$

Problem: Tail Recursion (6)

Historical Note:

- A paper “Bottom-Up Beats Top-Down for Datalog” from Jeffrey Ullman appeared in PODS’89.
- It proves that seminaive evaluation of the magic set transformed program is always at least as efficient as “top-down evaluation”.
- However, “top-down evaluation” as used in this paper is not SLD-resolution.

  It is a top-down query evaluation algorithm defined by Ullman himself (QRGT: Queue-Based Rule/Goal Tree Expansion). He even states that “this algorithm is easily seen to mimic the search performed by Prolog’s SLD resolution strategy”.

Problem: Tail Recursion (7)

Historical Note, continued:

- The paper contains a footnote that Prolog *implementations* usually contain a form of tail-recursion optimization that makes them faster than QRGT in certain cases.
- As shown here, it is not necessary to go down to the implementation level (e.g., the WAM). The efficient treatment of tail recursion is inherent in SLD-resolution.

Efficiency of Magic Sets (1)

Theorem:

- Let P be a rectified program.
- Let the standard left-to-right SIP-strategy and SLD selection function be used.
- Then for each magic fact $m\_p\_\beta(c_1, \ldots, c_k)$ that is derivable from $MAG(P) \cup EDB(P)$, there is a node in the SLD-tree with selected literal A, such that
$$magic[A] = m\_p\_\beta(c_1, \ldots, c_k).$$

Efficiency of Magic Sets (2)

Example:

- For each `m_path_bf`(i) there is a node `path`$(i,$ `X`$)$ in the SLD-tree.

Note:

- I.e. magic facts correspond to selected literals in the SLD-tree.
- Since both encode subqueries or predicate calls, there is a strong relation between both methods.

Efficiency of Magic Sets (3)

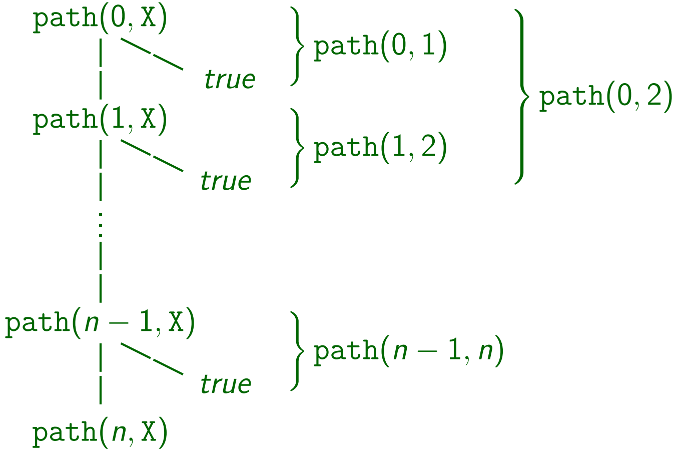
Definition:

- If a node $A \wedge B_1 \wedge \cdots \wedge B_n$ in the SLD-tree has a descendant node $(B_1 \wedge \cdots \wedge B_n)\theta$, where θ is the composition of the MGUs on this path, one says that $A\theta$ was proven as a lemma.

Theorem:

- Let P be a rectified program.
- Every non-magic fact about an IDB-predicate that is derivable from $MAG(P) \cup EDB(P)$ is proven in the SLD-tree as a lemma.

Efficiency of Magic Sets (4)



Efficiency of Magic Sets (5)

Reason for the Problem:

- With a linear number of nodes, one can prove a quadratic number of lemmas.

  The lemma depends on the path, or at least start and end node.

- The magic set method stores these lemmas explicitly. In the SLD-tree, they are only implicit.
- This problem occurs only for tail recursions.
- Otherwise (no tail recursions), the number of applicable rule instances in the transformed program is O(number of nodes in the SLD-tree).

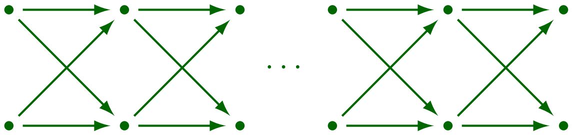
Efficiency of Magic Sets (6)

- If the program contains no function symbols and built-in predicates, bottom-up evaluation terminates and needs only polynomial time (wrt DB-size).
- SLD-resolution does not always terminate. Even if it does, it might need exponential time

  An example is given on the next slide.
- There are variants of SLD-resolution that use tabellation to avoid these problems (e.g. in XSB).
- But these methods have the same problem with tail recursions (they are equivalent to magic sets).

Efficiency of Magic Sets (7)

- There are exponentially many paths in this graph, and Prolog (SLD-resolution) follows them all:



- But the number of connected node pairs is quadratic, and magic sets compute only these.

  Because of join computations and duplicate elimination, the actual runtime is probably $O(n^2 * \log(n))$.

Contents

1 Introduction

2 Tail Recursion

3 Further Problems of Magic Sets

4 The SLDMagic Method

Introduction of Recursion (1)

- The “grandparent”-example on Slide 6 is non-recursive.
- However, the output of the magic set transformation for this example (Slide 7 and 8) is recursive:

$$
\begin{array}{lcl}
\texttt{parent}(\texttt{X}, \texttt{Y}) & \leftarrow & \texttt{m\_parent\_bf}(\texttt{X}) \wedge \\
 & & \texttt{mother}(\texttt{X}, \texttt{Y}). \\
\texttt{parent}(\texttt{X}, \texttt{Y}) & \leftarrow & \texttt{m\_parent\_bf}(\texttt{X}) \wedge \\
 & & \texttt{father}(\texttt{X}, \texttt{Y}). \\
\texttt{m\_parent\_bf}(\texttt{Y}) & \leftarrow & \texttt{m\_grandparent\_bf}(\texttt{X}) \wedge \\
 & & \texttt{parent}(\texttt{X}, \texttt{Y}).
\end{array}
$$

Introduction of Recursion (2)

- If the bottom-up machine cannot process recursive programs (e.g., a classical SQL-DBMS), this is a real problem.

  Otherwise, magic sets can be used for query optimzation in SQL-systems, when the query refers to views.

- Even if recursive programs can be processed, the recursion causes a significant overhead.

  Multiple relation variants are needed for the seminaive iteration, one cannot do a simple unfolding/expansion of the resulting relational algebra expressions, duplicate checks are necessary.

Introduction of Recursion (3)

- Why does this happen?
  - The problem is that there are two calls of the `parent`-predicate, and the magic set method uses only one predicate (magic set) to store the arguments (input values) of the calls.
  - The input values of the second call depend on the result values of the first call.
  - Since the magic set method does not distinguish between both calls, one gets a recursion.

Introduction of Recursion (4)

- Although this is syntactically a recursion, one can prove that a single application of the recursive rule about `m_parent_bf`, and two applications of the (recursive) rules about `parent` suffice.

  See evaluation sequence on the next page. The important point is that no new facts about `m_grandparent_bf` can be derived.

- No new facts will be derived if the recursive rules are iterated further.

- This is an example of a “bounded recursion”.

  If the bottom-up “machine” detects and optimizes bounded recursions, there is no problem.

Introduction of Recursion (5)

```
m_grandparent_bf(julia) ← true.
m_parent_bf(X)     ←  m_grandparent_bf(X).
parent(X, Y)       ←  m_parent_bf(X) ∧
                      mother(X, Y).
parent(X, Y)       ←  m_parent_bf(X) ∧
                      father(X, Y).
m_parent_bf(Y)     ←  m_grandparent_bf(X) ∧
                      parent(X, Y).
parent(X, Y)       ←  m_parent_bf(X) ∧
                      mother(X, Y).
parent(X, Y)       ←  m_parent_bf(X) ∧
                      father(X, Y).
```

Example: Web-Interface from Datalog

- In the next two examples, web queries will be written in Datalog.
- The following built-in predicates will be used:
  - `document(URL, Title, Text, Date)`
  - `link(From, To, Label)`
  - `index(Search_Term, URL, MaxResults)`
  - `server(URL, Server_Part)`
- Exercise: Which binding patterns can be supported with reasonable efficiency?

Context Switches Between Rules (1)

- Web pages that have changed since my last visit:

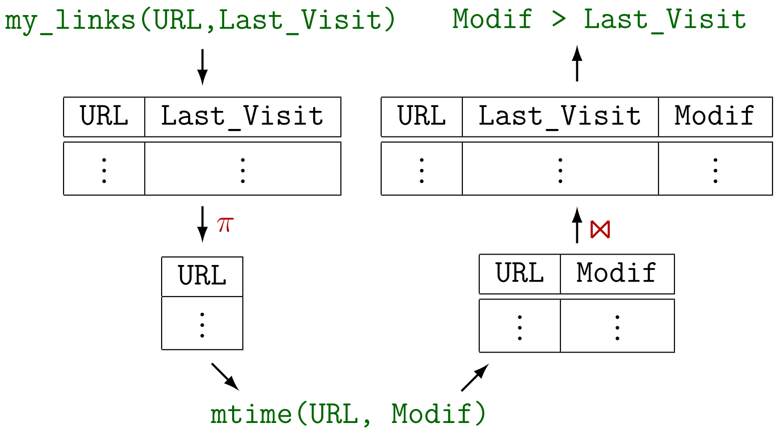
```
has_changed(URL)  :- my_links(URL, Last_Visit),
                     mtime(URL, Modif),
                     Modif > Last_Visit.

mtime(URL, Modif) :- document(URL, _, _, Modif).
```

- When the magic set for calling `mtime` is constructed, the bindings for `Last_Visit` are projected away.
- Later, these bindings must be reconstructed (with an expensive join) for evaluating `Modif > Last_Visit`.

Context Switches Between Rules (2)

- Getting results back into the context of the caller:



Context Switches Between Rules (3)

- Note that the source of the problem is again that different calls to a predicate are to be collected in a single magic predicate.
- Therefore, the context of the specific call must be forgotten when the input arguments are entered into the table for the magic predicate.
- This can also have advantages: Several identical calls are merged, the result is computed only once.
- In the example, if `URL` were not a key in `my_links`, the projection would eliminate duplicates.

Conditions for the Parameters (1)

- Magic sets pass to the called predicate only values for the parameters, e.g. `X = 5`, but not, e.g., `X > 5`.

```
has_changed(URL) :- my_links(URL, Last_Visit),
                    mtime(URL, Modif),
                    Modif > Last_Visit.
```

- Example query:

```
has_changed(URL), server(URL, 'www.pitt.edu').
```

- When the query is evaluated with magic sets,
  all pages in `my_links` are accessed.

  `has_changed(...)` must be evaluated first: `server(...)` needs `URL` bound.

Conditions for the Parameters (2)

- SLD resolution is more flexible: It would first replace the call to `has_changed` by its definition:

```
my_links(URL, Last_Visit),
mtime(URL, Modif),
Modif > Last_Visit,
server(URL, 'www.pitt.edu').
```

- Now `server(...)` can be evaluated directly after `my_links(...)`, before the expensive call `mtime(...)`.
- In this way, only pages of the server `'www.pitt.edu'` must be accessed.

Conditions for the Parameters (3)

- The problem is here that magic sets are bound to the predicate structure of the program.

  Which is again linked to the fact that identical calls to a predicate at different places in a program should be merged. This can be advantageous in certain situations and one cannot have both.

- SIP strategies determine the evaluation sequence only within a rule.

  Plus possibly bindings that are ignored when constructing the magic set.

- SLD selection functions determine the evaluation sequence within the entire remaining goal.

Conditions for the Parameters (4)

- Solutions for most of the above problems are known in the literature:
    - Magic sets with tail recursion optimization.
    - A version of magic sets that guarantees that non-recursive programs remain non-recursive.
    - A version of magic sets that can pass conditions like `X > 5` (unequalities) to the called predicate.
- My SLDMagic method (presented in the last part of this chapter) solves all of the above problems in a single framework.

Contents

1 Introduction

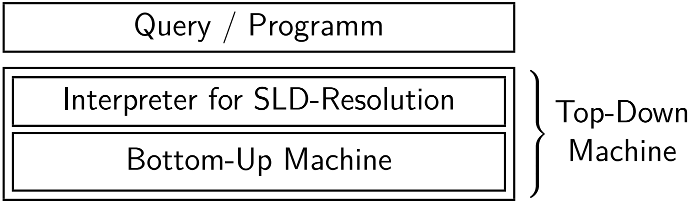
2 Tail Recursion

3 Further Problems of Magic Sets

4 The SLDMagic Method

The SLD-Magic Method

Starting Point (Meta-Interpreter):



Input Programm written as Datalog Facts (with lists):

```
rule(grandparent(X,Z),
     [parent(X,Y), parent(Y,Z)]).
```

Note:

- François Bry explained magic sets in this way.

The Meta-Interpreter (1)

- Initialization:

```
node(Query,[Query]) :- query(Query).
```

- SLD-Resolution:

```
node(Query, Child)  :- node(Query, [Lit|Rest]),
                       rule(Lit, Body),
                       append(Body, Rest, Child).
```

- Database Access:

```
node(Query, Rest)   :- node(Query, [Lit|Rest]),
                       db(Lit).
```

- Query is proven:

```
answer(Query)       :- node(Query, []).
```

The Meta-Interpreter (2)

Theorem (Simulation of SLD-Resolution):

- For each node $\mathcal{N}$ with goal $\leftarrow A_1 \wedge \cdots \wedge A_n$ in the SLD-tree there is a fact `node(`$Q\theta, [A'_1, \ldots, A'_n]$`)` derivable from the meta-interpreter, and a variable renaming σ, such that
  - $A'_i\sigma = A_i$ (for $i = 1, \ldots, n$), and
  - $Q\theta\sigma$ is the result of applying all MGUs on the path from the root to the node $\mathcal{N}$ to the query Q.
- And vice versa corresponds each derivable `node`-fact in this way to at least one node in the SLD-tree.

The Meta-Interpreter (3)

Definition:

- A program is at most tail-recursive iff for each rule $A \leftarrow B_1 \wedge \cdots \wedge B_n$ the predicates of B_i for $i \leq n-1$ do not depend on the predicate of A.

  I.e. only the last literal of every rule can be recursive.

Theorem (Termination):

- Let P be at most tail-recursive and let P, the DB, and Q be finite and without structured terms.
- Then bottom-up evaluation of the meta-interpreter terminates.

The Meta-Interpreter (4)

- Sometimes magic sets are better.
  Therefore the user should be able to choose the evaluation method for each body literal.
- This needs only two new rules in the interpreter.
- Start recursive call of SLD-resolution:

```
query(Lit)          :- node(_, [call(Lit)|_]).
```

- Use the recursively computed results:

```
node(Query, Rest) :- node(Query, [call(Lit)|Rest]),
                     answer(Lit).
```

The Meta-Interpreter (5)

- This is basically SLD resolution with tabulation:
    - The first rule puts the call into a table,
    - the second rule takes proven lemmas from a table in order to solve the literal.
- If `call(...)` is used for every body literal with an IDB predicate, one gets something very similar to magic sets with supplementary predicates.

    `query`-facts correspond to facts about magic predicates, `answer`-facts correspond to derived IDB-predicates, and `node`-facts correspond to facts about the supplementary predicates.

The Meta-Interpreter (6)

- With the possibility to select magic set behaviour, one can also overcome the termination problems of the pure SLD-meta-interpreter:
    - For every recursive call that is not tail-recursive, one uses: `call(...)`.
- For other body literals with IDB predicates, it is an intersting problem for the optimizer to choose between the two evaluation strategies.

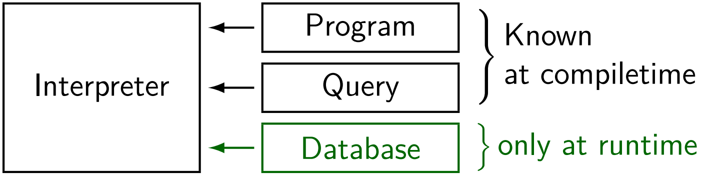
    It must try to find out how often the same call will be repeated. The strength of magic sets is that it avoids repeated calls (at the cost explained above).

Partial Evaluation (1)

    Interpreter
+ Partial Evaluator

    Compiler

Inputs for the Meta-Interpreter:



Partial Evaluation (2)

Idea:

- Fixpoint computation with conditional facts $\mathrm{F} \leftarrow \mathrm{C}$.

  C contains the part that is only known at runtime (typically bindings for the variables in F).

- Application of a rule gives
  - possibly a new conditional fact

    After the derivation step, variables in conditional facts are normalized (renamed to $X_0, X_1, \ldots$) to ensure that there are not unnecessarily many.
  - a specialized rule.

- At runtime, the evaluation works only with instances of the conditions.

Partial Evaluation (3)

Input Program:

$$\begin{array}{lcl} \mathtt{path}(\mathtt{X},\mathtt{Y}) & \leftarrow & \mathtt{edge}(\mathtt{X},\mathtt{Y}). \\ \mathtt{path}(\mathtt{X},\mathtt{Z}) & \leftarrow & \mathtt{edge}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{path}(\mathtt{Y},\mathtt{Z}). \end{array}$$

$?\,\mathtt{path}(0,\ \mathtt{X}).$

Initial Set of Conditional Facts:

- `query(path(0,X))` $\leftarrow$ `true.`
- `rule(path(X,Y),[edge(X,Y)])` $\leftarrow$ `true.`
- `rule(path(X,Z),[edge(X,Y), path(Y,Z)])` $\leftarrow$ `true.`
- `db(edge(X,Y))` $\leftarrow$ `edge(X,Y).`

Partial Evaluation (4)

Derivation Step:

Rule (from Meta-Int.):	$A \leftarrow B_1$	$\wedge\ B_2$
Conditional Facts:	$B'_1 \leftarrow C_1$	$B'_2 \leftarrow C_2$
MGU((B_1, B_2), (B'_1, B'_2)):	σ	
Part. eval. Rule:	$E \leftarrow C_1\sigma$	$\wedge\ C_2\sigma$
Conditional Fact:	$A\sigma \leftarrow E$	

Encoding of Result Literals:

E has the form $p(Y_1, \ldots, Y_n)$, where Y_i are those variables that appear in both, in $A\sigma$, and in one of the $C_i\sigma$.

Partial Evaluation (5)

Prototype: `http://www.informatik.uni-halle.de/~brass/sldmagic/`

Input:

```
path(X,Y)      :-  edge(X,Y).
path(X,Z)      :-  edge(X,Y), path(Y,Z).
?- path(0, X).
```

Output:

```
p0(X0)         :-  edge(0,X0).
p1(X1)         :-  edge(0,X1).
p0(X0)         :-  p1(X1), edge(X1,X0).
p1(X1)         :-  p1(X2), edge(X2,X1).
reach(d0,X0)   :-  p0(X0).
```

Partial Evaluation (6)

Conditional Facts:

```
db(edge(X0,X1)) :- edge(X0,X1).
rule(path(X0,X1),[edge(X0,X1)]) :- true.
rule(path(X0,X1),[edge(X0,X2),path(X2,X1)]) :- true.
query(path(0,X0)) :- true.
node(path(0,X0),[path(0,X0)]) :- true.
node(path(0,X0),[edge(0,X0)]) :- true.
node(path(0,X0),[edge(0,X1),path(X1,X0)]) :- true.
node(path(0,X0),[]) :- p0(X0).
node(path(0,X0),[path(X1,X0)]) :- p1(X1).
node(path(0,X0),[edge(X1,X0)]) :- p2(X1).
node(path(0,X0),[edge(X1,X2),path(X2,X0)]) :- p3(X1).
answer(path(X0,X1)) :- path(X0,X1).
```

Partial Evaluation (7)

Rules after partial evaluation:

-
  ```
  p0(X0) :- edge(0,X0).
  p1(X1) :- edge(0,X1).
  p2(X1) :- p1(X1).
  p3(X1) :- p1(X1).
  p0(X0) :- p2(X1), edge(X1,X0).
  p1(X1) :- p3(X2), edge(X2,X1).
  path(0,X0) :- p0(X0).
  ```
- In the version shown above already “copy rules” were eliminated.

  As can be seen, further optimizations are possible, but already this program does not more steps than SLD-resolution.