Deductive Databases and Logic Programming

(Winter 2007/2008)

# Chapter 7: Magic Sets

- SIP-Strategies (“**S**ideways **I**nformation **P**assing”)
- Adorned Program, Magic Predicates
- Correctness and Efficiency
- Problems and Improvements





## Objectives

After completing this chapter, you should be able to:

- perform the magic set transformation for a given input program (and explain how it works)
- name different SIP strategies
- compare magic sets with SLD resolution
- name some problems of the magic set method and sketch possible solutions.





## Overview







## Introduction (1)

Example (Grandparents of Julia):

- Logic Program (IDB-Predicates and Query):

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

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

# Introduction (2)

Problem:

- Naive/Seminaive Bottom-Up Evaluation computes all parent- and grandparent-relationships of all persons in the database.
- Until now, the actual query is considered only at the very end of query evaluation — after the entire minimal model was computed.
- Therefore, the method is not goal-directed: It computes many superfluous facts, which are not relevant for the query.





# Introduction (3)

Solution:

- The "Magic Set" Transformation rewrites the program such that the rules can only "fire" when their result (the derived fact) is relevant for the query.
- This is done by making the occurring queries and subqueries explicit. They are encoded as facts of "magic predicates". E.g. the query

  $$?\ \texttt{grandparent}(\texttt{julia}, \texttt{X})$$

  is represented as

  $$\texttt{m\_grandparent\_bf}(\texttt{julia}).$$





# Introduction (4)

More About Encoding of Queries:

- Consider again the correspondance:
  - ◇ Query: $?\ \texttt{grandparent}(\texttt{julia}, \texttt{X})$
  - ◇ Magic Fact: $\texttt{m\_grandparent\_bf}(\texttt{julia})$.
- Magic facts should be representable in a database, and therefore should not contain variables.
- Solution: The binding pattern indicates the position of the variables (their name is not important).
- Only the values of constants in the query (bound arguments) are explicitly stored in the magic fact.





# Introduction (5)

Example Output, First Part:

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

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

# Introduction (6)

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 must be represented as a magic fact 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`.





# Introduction (7)

Summary (Equivalence):

- The "Magic Set" transformation produces a program which is equivalent for the given query:
  - ◇ The extension of the predicate "`answer`" in the minimal model of the transformed program is the same as in the minimal model of the original program.
- But often the minimal model of the transformed program is much smaller than the minimal model of the original program.

  It contains only IDB-facts that are relevant for the given query.





# Introduction (8)

Summary (Equivalence), Continued:

- This equivalence (for the predicate `answer`) is independend of the extensions of the EDB-predicates:
  - ◇ No database access is needed during the transformation.

    Therefore, the transformation itself is quite efficient (one usually assumes that external memory accesses are expensive).
  - ◇ The transformed program can be executed several times, even when the database state was changed in the meantime.





# Introduction (9)

Input Program
(simple, but not efficient)

⇓

"Magic Set" Transformation

⇓

Output Program
(returns the same answers,
but is evaluated more efficiently)

# Introduction (10)

Magic Sets and Built-in Predicates:

- The magic set transformation can also improve the termination of programs with built-in predicates:
  - ◇ E.g., `append` has an infinite extension.
  - ◇ But for a concrete query, only a finite number of facts might be needed.

    E.g., when two given lists are appended.
  - ◇ In this way, the magic set transformation might turn an infinite minimal model into a finite one.

    Of course, it depends on the program and the query whether this works.





# Introduction (11)

Top-Down vs. Bottom-Up:

- Before the magic set transformation, there were two competing evaluation approaches:
  - ◇ Top-down evaluation (e.g. SLD-resolution): Starts from the query, simplifies it, until facts can be used. Advantage: Goal-directed.
  - ◇ Bottom-up evaluation (e.g., seminaive method): Starts from the facts, computes derived facts, until answers to the query are reached. Advantage: Avoids duplicate work, ensures termination.
- Magic sets combine the advantages of both.





# Introduction (12)

Magic Sets — A Source Code Level Transformation:

- The underlying "bottom-up machine" can remain unchanged.
- It is always good to separate problems and solve them one after the other.
- One can understand the method on a high level of abstraction (Herbrand models instead of internal data structures).
- However, it is probably advantageous for an implementation to treat the magic predicates specially.





# Introduction (13)

IDB predicates as procedures:

- With magic sets, IDB predicates can be understood as procedures with input and output arguments:
  - ◇ Input: Relation $M$ with bindings for the input arguments (this is the "Magic Set").
  - ◇ Output: Relation $R$ for all arguments.

    It does not suffice to return only a relation for the output arguments because the connection to the input arguments would not be clear if $M$ contains more than one tuple.
  - ◇ If $E$ is the original extension of the predicate, $R = E \ltimes M$ holds (this is a semi-join).

# Introduction (14)

IDB predicates as procedures, continued:

- Example:

$M$:

| m_parent_bf |
|---|
| emil |
| frida |

$\longrightarrow$ Procedure: parent_bf $\longrightarrow$

$R$:

| parent_bf | |
|---|---|
| emil | arno |
| emil | birgit |
| frida | chris |
| frida | doris |

The database might contain many more mother/father relationships, but only the required parent tuples are derived.

- However, for recursive predicates, $M$ might still be extended later.





# Overview







# SIP-Strategies (1)

Motivation:

- SIP = „Sideways Information Passing"

  When rules are evaluated, information is passed "sideways": from a body literal that is evaluated earlier to one that is evaluated later.

- $\texttt{grandparent(X,Z)} \leftarrow \texttt{parent(X,Y)} \land \texttt{parent(Y,Z)}$.
  - ◇ Variable binding from the caller: $\texttt{X} = \texttt{julia}$.
  - ◇ This is passed to the first body literal.
  - ◇ By evaluating this body literal, one gets bindings for Y, e.g. $\texttt{Y} = \texttt{emil}$ and $\texttt{Y} = \texttt{frida}$.
  - ◇ These are passed to the second body literal.





# SIP-Strategies (2)

- However, for the query "? grandparent(X, arno)", it is more efficient to start the evaluation of

  $$\texttt{grandparent(X,Z)} \leftarrow \texttt{parent(X,Y)} \land \texttt{parent(Y,Z)}$$

  with the second body literal.
  - ◇ Then the binding $\texttt{Z} = \texttt{arno}$ can be used.
  - ◇ This gives bindings for Y, which can be passed to the first body literal.
- If instead one evaluates the first body literal first, this is done with the binding pattern ff, and one has to compute the complete extension of parent.

# SIP-Strategies (3)

Definition:

- Given a rule $A \leftarrow B_1 \wedge \cdots \wedge B_m$ and a binding pattern $\beta$ for $pred(A)$,
- a SIP-strategy defines an evaluation sequence for the body literals, i.e. a permutation

$$\pi: \{1, \ldots, m\} \rightarrow \{1, \ldots, m\},$$

- and for every $k \in \{1, \ldots, m\}$ a valid binding pattern $\beta_{\pi(k)} \in valid\big(pred(B_{\pi(k)})\big)$ such that

$$input(B_{\pi(k)}, \beta_{\pi(k)}) \subseteq input(A, \beta) \cup \bigcup_{j=1}^{k-1} vars(B_{\pi(j)}).$$





# SIP-Strategies (4)

Note:

- This is the same condition for $\pi$ and $\beta_{\pi(k)}$ as in the definition of "range restricted rule".
- A given evaluation sequence determines "maximally bound" binding patters for the body literals:
  - ◇ Values for variables in "bound" argument positions in the head literal are known.
  - ◇ Values for variables in body literals that were evaluated earlier are known.
  - ◇ All other variables do not have a known value yet, thus they lead to "free" argument positions.





# SIP-Strategies (5)

- Most SIP-strategies choose the above "maximal" binding pattern that uses all existing bindings.
- Therefore, the real decision is the evaluation sequence for the body literals. The binding patterns are then often automatically determined.

  However, the possible evaluation sequences depend on the valid binding patterns for the body literal: Some predicates can only be evaluated if certain arguments are bound.

- A SIP-strategy can ignore existing bindings and choose a more general binding pattern.

  Possible reasons are explained on the next slide.





# SIP-Strategies (6)

Reasons for not choosing the maximal binding pattern:

- Not all binding patters might be implemented.

  This is obvious for built-in predicates, but happens also for IDB-predicates in separately compiled modules (modules define "exported binding patterns").

- One needs the more general binding pattern anyway at some other place in the program.

  In this way, one avoids duplicating the rules. But unless the other binding pattern is `ff...f`, one computes more tuples (if calls disjoint).

- to simplify the magic rules.

  An even more general kind of "SIP-strategy" permits to choose a subset of the earlier evaluated body literals for the magic rule.

# SIP-Strategies (7)

Exercise:

- Consider this rule (with only IDB predicates):

```
ship_to(ProdName, City) ←
            has_ordered(CustNo, ProdNo) ∧
            customer_city(CustNo, City) ∧
            product_name(ProdNo, ProdName).
```

- Select a good evaluation sequence for each of the following calls, and state the binding patterns:
  - ◇ `ship_to(X, 'Halle')`.
  - ◇ `ship_to('Van Tastic', X)`.
  - ◇ `ship_to('Van Tastic', 'Halle')`.





# SIP-Strategies (8)

Framework for SIP-Strategies:

- Let $A \leftarrow B_1 \wedge \cdots \wedge B_m$ be called with binding pattern $\beta$. One chooses first $\pi(1)$ (i.e. the first body literal to evaluate), then $\pi(2)$, and so on.
- Given that one has already chosen $\pi(1), \ldots, \pi(k)$, the $i$-th body literal is possible with binding pattern $\beta' \in \mathit{valid}\big(\mathit{pred}(B_i)\big)$ iff the literal has not been chosen yet, i.e. $i \in \{1, \ldots, m\} - \{\pi(1), \ldots, \pi(k)\}$ and

$$\mathit{input}(B_i, \beta') \subseteq \mathit{input}(A, \beta) \cup \bigcup_{j=1}^{k} \mathit{vars}(B_{\pi(j)}).$$





# SIP-Strategies (9)

Example:

- Consider the following rule is called with $p(X, 3)$:

$$p(X, Y) \leftarrow X < Y \wedge q(X).$$

- At the beginning, only the second body literal is evaluable (with binding pattern `f`).

  The only valid binding pattern for $<$ is `bb`, therefore the first body literal cannot be evaluated at this point (although it has more bound arguments than $q(X)$: the value for $Y$ is already known).

- Thus, all SIP-strategies must select $\pi(1) = 2$.
- After $q(X)$ is evaluated, the value of $X$ is known, and the first literal becomes evaluable: $\pi(2) = 1$.





# SIP-Strategies (10)

Common SIP-Strategies:

- Among all possible $(i, \beta)$, choose one such that $\beta$ has the smallest number of free argument positions.
- Among all possible $(i, \beta)$, choose one such that $\beta$ has the largest number of bound arguments.
- Among all possible $(i, \beta)$, choose one such that $i$ is minimal, and among those one such that $\beta$ has the largest number of bound argument positions.

  This strategy evaluates body literals in the sequence given by the programmer as far as possible.

# SIP-Strategies (11)

Example:

- Consider the rule

  $$p(X_1, X_2) \leftarrow q(X_1, Y) \wedge r(X_1, X_2, Z_1, Z_2)$$

  and the call $p(a, b)$.
- A SIP-strategy that tries to maximize the number of bound argument positions begins the evaluation with the second body literal: $r(X_1, X_2, Z_1, Z_2)$.
- A SIP-strategy that minimizes the number of free argument positions chooses $q(X_1, Y)$ first.





# SIP-Strategies (12)

Further Input for SIP-Strategies:

- The SIP-strategy is an important component of the optimizer of a deductive DBMS.
- It should take also the following information into account:
  - ◇ Keys
  - ◇ Indexes
  - ◇ Size of Relations
  - ◇ Number of different values in an attribute





# SIP-Strategies (13)

Goals of SIP-Strategies:

- Try to keep intermediate results small.
- "Fail as early as possible".
- Call expensive predicates only when cheap tests were successful. Expensive predicates are:
  - ◇ Recursive predicates
  - ◇ Predicates that need themselves many joins and possibly a duplicate elimination.
  - ◇ Built-in predicates with complicated computations or slow network accesses.





# SIP-Strategies (14)

Exercises:

- $p(X, Z) \leftarrow q(X, Y_1, Y_2, Y_3) \wedge r(Y_1, Z)$, with call $p(a, b)$, when the first argument of $q$ and $r$ is a key.
- Suppose $p$ is defined by $p(X) \leftarrow q_1(X) \wedge q_2(X)$ and is called with binding pattern `f`. Cost estimates:
  - ◇ $q_1$:`f` produces 100 tuples in 100 ms.
  - ◇ $q_1$:`b` checks a single value in 3 ms (index).
  - ◇ $q_2$:`f` produces 1000 tuples in 200 ms.
  - ◇ $q_2$:`b` checks a single value in 100 ms (FT scan).
  - ◇ Sorting/intersecting the two sets costs 1000 ms.

# Overview







# Adorned Program (1)

Goal:

- The first step of the transformation is to make the binding patterns of the IDB-predicates explicit.

  This simplifies the definition of the main magic set transformation.

- I.e. the predicate $p$ is renamed to $p\_\beta$. Several versions of a predicate are produced for different $\beta$.

  Sometimes one IDB-predicate is called with different binding patterns.

- Furthermore, the body literals are ordered in the evaluation sequence.

  In this way, the information from the SIP-strategy is encoded in the program.





# Adorned Program (2)

Definition:

- Let a logic program $P$, valid binding patterns $valid$ and a SIP-strategy $\mathcal{S}$ be given.
- For each predicate $p \in \mathcal{P}_{\mathcal{IDB}}(P)$, $p \neq \texttt{answer}$, and every binding pattern $\beta \in valid(p)$ a new predicate $p\_\beta$ is introduced.
- For a literal $A = p(t_1, \ldots, t_n)$ and binding pattern $\beta \in valid(p)$ let

$$adorn(A, \beta) := \begin{cases} p\_\beta(t_1, \ldots, t_n) & \text{if } p \in \mathcal{P}_{\mathcal{IDB}}(P), \\ & \quad p \neq \texttt{answer} \\ p(t_1, \ldots, t_n) & \text{otherwise.} \end{cases}$$





# Adorned Program (3)

Definition, continued:

- The program $AD(P)$ contains for each rule

$$A \leftarrow B_1 \wedge \cdots \wedge B_m \qquad \text{from } IDB(P)$$

and each binding pattern $\beta \in valid\big(pred(A)\big)$ the rule

$$\begin{aligned} adorn(A,\, \beta) \ \leftarrow\ & adorn(B_{\pi(1)},\, \beta_{\pi(1)}) \ \wedge \cdots \wedge \\ & adorn(B_{\pi(m)},\, \beta_{\pi(m)}), \end{aligned}$$

where $\pi$ and $\beta_1, \ldots, \beta_m$ are determined by the SIP-strategy $\mathcal{S}$.

# Adorned Program (4)

Note:

- It is theoretically simpler to process every rule for each possible binding pattern.
- The unnecessary binding patterns are eliminated when one later deletes all predicates (and their definitions) on which "`answer`" does not depend.
- In practice, only necessary $p\_\beta$ are constructed.

  E.g. one manages a set of all combinations $(p, \beta)$ that still have to be processed. This is intialized with $\{(\texttt{answer}, \texttt{f}\ldots\texttt{f})\}$. In each step, one takes an element from this set and generates the rules for $p\_\beta$. If the rule bodies contain new combinations of predicates and binding patterns, one inserts them into the set.





# Adorned Program (5)

- Binding patterns chosen by the SIP-strategy for EDB-/builtin predicates are important to determine which index/implementation variant is to be used.
- But the names of these predicates are determined in the database/the system, they cannot be changed.

  In contrast, names of IDB-predicates ($\neq$ `answer`) are only important within the program.
- Furthermore, the explicit binding patterns are a preparation for the magic set transformation. This is not useful for EDB-predicates, because their complete extensions are already stored.





# Adorned Program (6)

Exercise:

- Compute $AD(P)$ (as far as relevant for `answer`). Use a good SIP-strategy:
  - ◇
    $$\begin{array}{lcl} \texttt{parent(X,Y)} & \leftarrow & \texttt{mother(X,Y)}. \\ \texttt{parent(X,Y)} & \leftarrow & \texttt{father(X,Y)}. \\ \texttt{grandparent(X,Z)} & \leftarrow & \texttt{parent(X,Y)} \wedge \texttt{parent(Y,Z)}. \\ \texttt{answer(X)} & \leftarrow & \texttt{grandparent(julia,X)}. \end{array}$$
  - ◇
    $$\begin{array}{lcl} \texttt{append([], L, L)} & \leftarrow & \texttt{true}. \\ \texttt{append(X, L, Y)} & \leftarrow & \texttt{cons(H,T,X)} \wedge \texttt{cons(H,TL,Y)} \wedge \\ & & \texttt{append(T, L, TL)}. \\ \texttt{answer(X)} & \leftarrow & \texttt{append([a], [b, c], X)}. \end{array}$$





# Adorned Program (7)

Exercise, continued:

- Compute $AD(P)$ (as before):
  - ◇ „Same Generation Cousins“:
    $$\begin{array}{lcl} \texttt{sg(X,X)} & \leftarrow & \texttt{person(X)}. \\ \texttt{sg(X,Y)} & \leftarrow & \texttt{parent(X,Xp)} \wedge \texttt{parent(Y,Yp)} \wedge \\ & & \texttt{sg(Xp,Yp)}. \\ \texttt{answer(X)} & \leftarrow & \texttt{sg(julia,X)}. \end{array}$$
  - ◇ Abstract Example:
    $$\begin{array}{lcl} \texttt{answer(yes)} & \leftarrow & \texttt{p(a,b)}. \\ \texttt{p(X,Z)} & \leftarrow & \texttt{q(X,Y)} \wedge \texttt{q(Y,Z)}. \\ \texttt{q(X,Y)} & \leftarrow & \texttt{r(a,X,Y)}. \end{array}$$

# Adorned Program (8)

Lemma:

- Let $\mathcal{I}$ be the minimal model of $P$ and $\mathcal{I}'$ be the minimal model of $AD(P) \cup EDB(P)$.
- For every $p \in \mathcal{P}_{\mathcal{IDB}}(P)$, $p \neq$ `answer`, and every $\beta \in valid(p)$:

$$\mathcal{I}'[\![p\_\beta]\!] = \mathcal{I}[\![p]\!].$$

- Furthermore: $\mathcal{I}'[\![\texttt{answer}]\!] = \mathcal{I}[\![\texttt{answer}]\!]$.

I.e. this part of the transformation has not changed the minimal model in an important way. It only renamed predicates (and possibly duplicated them).





# Magic Set Transformation (1)

Notation:

- For a literal $A = p\_\beta(t_1, \ldots, t_n)$ let

$$magic[A] := m\_p\_\beta(t_{i_1}, \ldots, t_{i_k}),$$

where $1 \leq i_1 < \cdots < i_k \leq n$ are the argument positions with $\beta(i_j) = \texttt{b}$.

- For a literal $A = \texttt{answer}(\texttt{X}_1, \ldots, \texttt{X}_n)$ with the special predicate `answer` let $magic[A] := true$.

Example:

- $magic[\texttt{parent\_bf}(\texttt{X},\texttt{Y})] := \texttt{m\_parent\_bf}(\texttt{X})$.





# Magic Set Transformation (2)

Definition:

- Let $P = EDB(P) \cup IDB(P)$ be a logic program and $AD(P)$ be the version of $IDB(P)$ with explicit binding patterns.
- Then $MAG(P)$ contains the following rules:
  - ◇ For each rule $A \leftarrow B_1 \wedge \cdots \wedge B_m$ in $AD(P)$:

$$A \leftarrow magic[A] \wedge B_1 \wedge \cdots \wedge B_m.$$

  - ◇ For each rule $A \leftarrow B_1 \wedge \cdots \wedge B_m$ in $AD(P)$ and every $i \in \{1, \ldots, m\}$ with $pred(B_i) \in \mathcal{P}_{\mathcal{IDB}}(P)$:

$$magic[B_i] \leftarrow magic[A] \wedge B_1 \wedge \cdots \wedge B_{i-1}.$$





# Magic Set Transformation (3)

Definition:

- Rules of the type

$$A \leftarrow magic[A] \wedge B_1 \wedge \cdots \wedge B_m$$

are called "modified rules".

- Rules of the type

$$magic[B_i] \leftarrow magic[A] \wedge B_1 \wedge \cdots \wedge B_{i-1}$$

are called "magic rules".

- A "magic fact" is a fact (ground atom) of the form $m\_p\_\beta$. All other facts are called "non-magic facts".

# Magic Set Transformation (4)

Exercises:

- Check that the defined "magic set" transformation gives the result on Slide 7-8 and 7-9 (without suffix `bf`) for the "grandparent" example (Slide 7-4).
- Compute the result of the transformation for `append` with binding pattern `bbf`.

```
append([], L, L)  ←  true.
append(X, L, Y)   ←  cons(H,T,X) ∧ cons(H,TL,Y) ∧
                     append(T, L, TL).
answer(X)         ←  append([a], [b, c], X).
```





# Correctness (1)

Lemma:

- Let $\mathcal{I}_{AD}$ be the minimal model of $AD(P) \cup EDB(P)$, $\mathcal{I}_{MAG}$ be the minimal model of $MAG(P) \cup EDB(P)$.
- For all non-magic facts $A$ the following holds:
  - ◇ If $\mathcal{I}_{MAG} \models A$, then $\mathcal{I}_{AD} \models A$.

    Proof Sketch: Induction on the number of derivation steps. A non-magic fact can only be derived by a "modified rule", but this is only a restricted version of the corresponding rule in $AD(P)$.
  - ◇ If $\mathcal{I}_{AD} \models A$ and $\mathcal{I}_{MAG} \models magic[A]$, then $\mathcal{I}_{MAG} \models A$.

    Proof Sketch: Induction on the number of derivation steps of $A$ from $\mathcal{I}_{AD}$.





# Correctness (2)

Theorem:

- Let $\mathcal{I}_{AD}$ and $\mathcal{I}_{MAG}$ be as above, and $\mathcal{I}$ be the minimal model of $EDB(P) \cup IDB(P)$. Then the following holds:

$$\mathcal{I}_{MAG}[\![\mathtt{answer}]\!] = \mathcal{I}_{AD}[\![\mathtt{answer}]\!] = \mathcal{I}[\![\mathtt{answer}]\!].$$

- I.e. the transformed program is equivalent to the original program in the sense that it returns the same answers.

  The two programs are not logically equivalent. Actually, that is not even defined, because the programs are based on different signatures. The programs could be called "answer-equivalent".





# Correctness (3)

Theorem:

- Let $P$ be range-restricted with respect to $valid$.
- Then $MAG(P)$ is range-stricted with respect to $valid'$, where

$$valid'(q) := \begin{cases} \{\mathtt{f}\ldots\mathtt{f}\} & \text{if } q \text{ has the form } p\_\beta/m\_p\_\beta \\ valid(q) & \text{otherwise.} \end{cases}$$

- I.e. the transformed program can be evaluated by iteration of the $T_P$-Operator.

# Overview







# Supplementary Predicates (1)

Example:

- Rule from $AD(P)$, all $B_i$ with IDB-predicates:

$$A \leftarrow B_1 \wedge B_2 \wedge B_3.$$

- Result of the magic set transformation:

$$\begin{array}{lll} magic[B_1] & \leftarrow & magic[A_1]. \\ magic[B_2] & \leftarrow & magic[A_1] \wedge B_1. \\ magic[B_3] & \leftarrow & magic[A_1] \wedge B_1 \wedge B_2. \\ A & \leftarrow & magic[A_1] \wedge B_1 \wedge B_2 \wedge B_3. \end{array}$$

- The same joins and selections are computed several times.





# Supplementary Predicates (2)

Solution:

- Compute each join only once, store the result in a new "supplementary predicate":

$$\begin{array}{lll} magic[B_1] & \leftarrow & magic[A_1]. \\ S_1 & \leftarrow & magic[A_1] \wedge B_1. \\ magic[B_2] & \leftarrow & S_1. \\ S_2 & \leftarrow & S_1 \wedge B_2. \\ magic[B_3] & \leftarrow & S_2. \\ A & \leftarrow & S_2 \wedge B_3. \end{array}$$

- The arguments of the $S_i$ are those variables from $magic[A] \wedge B_1 \wedge \cdots \wedge B_i$, that are still needed, i.e. that occur in $B_{i+1}, \ldots, B_m, A$.





# Supplementary Predicates (3)

- The deductive DBMS CORAL uses this method ("Magic Sets with Supplementary Predicates").

CORAL is installed on our SUN machines. Before calling it, one must use `setenv CORALROOT /usr/central/coral`. Then it can be called with `$CORALROOT/bin/coral`. A manual is in `$CORALROOT/doc/manual.ps` (use `gv` to display it). Homepage: `http://www.cs.wisc.edu/coral/`.

In CORAL, rules are written into modules, an example `sg.P` is shown on the next slide. Facts are written outside modules, e.g. into `*.F`-files. The files are processed with, e.g., `consult(sg.P)`. For modules, this does the magic set transformation, the result is stored in `sg.P.M` (quite readable, i.e. one can look at the result of the transformation).

Strings are written, e.g., `"abc"`. If it has the form `[a-z][a-zA-Z0-9_.]*`, no `"` is needed. Computation: `Y = X+1`. Query syntax: `? sg(julia, X)`. To get all answers immediately: `clear(interactive_mode)`. Also useful: `help.`, `quit.`, `list_rels.`

# Supplementary Predicates (4)

Example (CORAL):

- Rules in Coral are written in modules, with exported predicates and possible binding patterns defined.

```
module same_generation_cousins.

export sg(bf).

sg(X,X) :- person(X).
sg(X,Y) :- parent(X,Xp), sg(Xp,Yp), parent(Y,Yp).

end_module.
```

`export` also works with several binding patterns, e.g. `sg(bf,ff)`. More bound arguments in the call are possible, but not so efficient.





# Supplementary Predicates (5)

- Example:

$$\mathtt{sg(X,Y)} \leftarrow \mathtt{parent(X,Xp)} \wedge \mathtt{sg(Xp,Yp)} \wedge \mathtt{parent(Y,Yp)}.$$

- Result (if `parent` is an EDB-predicate):

$$\begin{array}{lcl} \mathtt{sup\_2\_1(X,Xp)} & \leftarrow & \mathtt{m\_sg\_bf(X)} \wedge \\ & & \mathtt{parent(X,Xp)}. \\ \mathtt{m\_sg\_bf(X)} & \leftarrow & \mathtt{sup\_2\_1(X,Xp)}. \\ \mathtt{sg\_bf(X,Y)} & \leftarrow & \mathtt{sup\_2\_1(X,Xp)} \wedge \\ & & \mathtt{sg\_bf(Xp,Yp)} \wedge \\ & & \mathtt{parent(Y,Yp)}. \end{array}$$

- In CORAL, the $i$-th supplementary predicate of the $n$-th rule is named "$sup\_n\_i$".





# Supplementary Predicates (6)

- Example (continued):

$$\mathtt{sg(X,X)} \leftarrow \mathtt{person(X)}.$$

- The transformation of this non-recursive rule that only uses an EDB-predicate is easy:

$$\mathtt{sg\_bf(X,X)} \leftarrow \mathtt{m\_sg\_bf(X)} \wedge \mathtt{person(X)}.$$

- Coral also supports other transformations.

  Try one of: "`@magic+.`", "`@sup_magic+.`" (this is the default), "`@factoring+.`", "`no_rewriting+.`", "`sup_magic_indexing+.`".

- The SIP-strategy is left-to-right.





# Supplementary Predicates (7)

Note:

- In this method, magic sets are always directly derived from supplementary predicates.
- One can try to replace the magic predicates by the supplementary predicates.
- If a magic predicate is defined by only one rule (only one call of $p\_\beta$), this is simply a macro-expansion.
- Otherwise, one would have to duplicate rules.
- Depending on the application, it might be an advantage to distinguish different calls of a predicate.

# Rectification (1)

- Problem/Example:

$$p(Y_1, Y_2, Y_3) \leftarrow q(X, X, Y_1, Y_2, Y_3).$$
$$q(a, b, Y_1, Y_2, Y_3) \leftarrow r(Y_1) \land r(Y_2) \land r(Y_3).$$

- The basic magic set method as explained above distinguishes only between "bound" and "free" argument positions.
- It calls $q$ with the binding pattern `fffff`.
- Suppose that there are $n$ facts about $r$. Then $n^3$ facts about $q$ will be derived, although the rule does not match the call.





# Rectification (2)

- Since the body of the first rule is not unifiable with the head of the second rule, SLD-resolution would immediately stop (without looking at the $r$-facts).

  Such a situation probably does not happen often in practice. But since one wants to prove that magic sets are (in some sense) as efficient as (or really as goal-directed as) SLD-resolution, this is a problem.

- The magic set method can pass only concrete values for the arguments to a called predicate.
- The basic method cannot pass the information to $q$ that the first two arguments must be equal.





# Rectification (3)

Definition:

- A logic program is rectified iff no body literal contains the same variable more than once, i.e. for every body literal $p(t_1, \ldots, t_n)$, if $t_i = t_j$ for $i \neq j$, then $t_i$ is a constant.

Remark:

- In the magic predicates, free argument positions are projected away (not represented). If the program is rectified, this does not lead to a loss of information.





# Rectification (4)

- Every logic program that does not contain function symbols (structured terms) can be transformed into an equivalent, rectified program.

  Again, equivalent means that it produces the same answer.

- The rectification is done by introducing predicate variants that contain at different argument positions the same arguments: $p^{(i_1,\ldots,i_n)}(t_1, \ldots, t_k)$ corresponds to $p(t_{i_1}, \ldots, t_{i_n})$, e.g.
  - ◇ $q^{(1,1,2,3,4)}(\mathtt{X}, \mathtt{Y}_1, \mathtt{Y}_2, \mathtt{Y}_3)$ means $q(\mathtt{X}, \mathtt{X}, \mathtt{Y}_1, \mathtt{Y}_2, \mathtt{Y}_3)$.

# Rectification (5)

- One specializes the rules of the original predicate once for each such predicate variant:
  - ◇ E.g., the critical rule $q(a, b, \ldots) \leftarrow \ldots$ is deleted in the specialization for the predicate variant $q^{(1,1,2,3,4)}$.

    Note that this is a predicate name. In practice, one of course has to encode it without exponent, parentheses, commas.
- In general, one tries to unify the rule head with $p(t_{i_1}, \ldots, t_{i_n})$. If it is unifiable, the result is encoded with the new predicates: $p^{(i_1,\ldots,i_n)}$ in the head, and in the body as needed to ensure rectification.





# Rectification (6)

- As explained above, rectification is applied before the adorned program is computed and the SIP-strategy is applied.
- However, one could do the rectification also together with the adornment.
- Then binding patterns consist not of `b` and `f`, but of "constant", "variable-1", "variable-2", and so on.

  Note that it is no problem if a bound variable appears twice in a body literal. So one might need fewer predicate variants in this way. Note also that if one wants to come close to SLD-resolution, only SIP-strategies are interesting that do not ignore existing bindings.





# Overview







# Problem: Tail Recursion (1)

Example:

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

$$path(X, Y) \leftarrow link(X, Y).$$
$$path(X, Z) \leftarrow link(X, Y) \land path(Y, Z).$$
$$?\ path(\text{"http://.../0"},\ X).$$

- EDB-Relation (Hypertext-Graph):

$$link := \{(i-1, i) \mid 1 \le i \le n\}.$$

$$0 \longrightarrow 1 \longrightarrow \cdots \longrightarrow n$$

# Problem: Tail Recursion (2)

Complexity of Magic Sets:

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

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

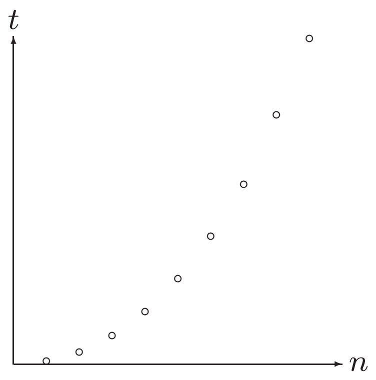
- These are $(n+2)(n+1)/2$ facts, thus the runtime is at least quadratic (probably more).
- This example should run in linear time!





# 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 $link$: $O\big(\log(n)\big)$.
- Total complexity: $O\big(n * \log(n)\big)$.
- If one could access $link\_\mathtt{bf}$ in time $O(1)$ (e.g., hash table), the total runtime would really be linear.





# Problem: Tail Recursion (5)

$$path(0, X)$$

$$link(0, X) \qquad link(0, Y) \wedge path(Y, X)$$

$$\mathtt{X} = 1 \qquad path(1, X)$$

$$\vdots$$

$$path(n, X)$$

$$link(n, X) \qquad link(n, Y) \wedge path(Y, X)$$

# 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 $\theta$ is the composition of the MGUs on this path, one says that $A\theta$ was proven as a lemma.

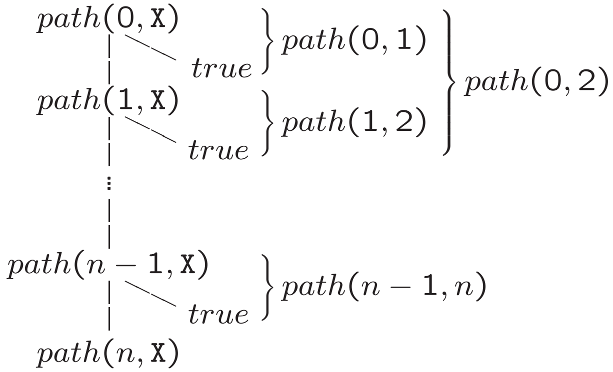
Theorem:

- 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)

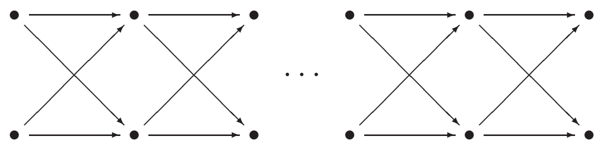
- If the program contains no function symbols and built-in predicates, bottom-up evaluation terminates and needs only polynomial time (wrt DB-size).
- SLD-resolution does not always terminate. Even if it does, it might need exponential time

  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 $O(n^2 * \log(n))$.





# Further Problems (1)

Nonrecursive Programs can Become Recursive:

- The “grandparent”-example on Slide 7-4 is nonrecursive.
- However, the output of the magic set transformation for this example (Slide 7-8 and 7-9) is recursive:

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





# Further Problems (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.





# Further Problems (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.

# Further Problems (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.





# Further Problems (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).
```





# Further Problems (6)

Example:

- 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?





# Further Problems (7)

Context Switches:

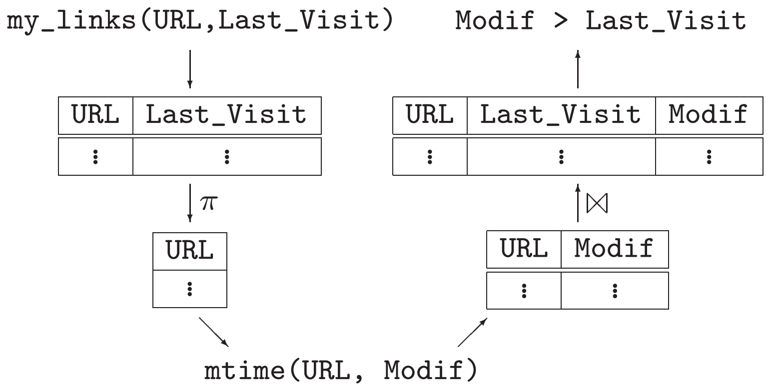
- Web pages that have changed since my last visit:

```
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.

# Further Problems (8)

- Getting results back into the context of the caller:








# Further Problems (9)

- 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.





# Further Problems (10)

Conditions for the Parameters:

- 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.





# Further Problems (11)

- 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.

# Further Problems (12)

- 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.





# Further Problems (13)

- 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.





# Overview

1. Introduction (Examples, Motivation, Idea)
2. SIP Strategies, Adorned Program
3. The Magic Set Transformation
4. Improvements
5. Efficiency Comparison with SLD-Resolution
6. SLDMagic Method





# The SLD-Magic Method

Starting Point (Meta-Interpreter):

| |
|---|
| Anfrage / Programm |
| Interpreter for SLD-Resolution |
| Bottom-Up Machine |

Top-Down Machine

Input Programm written as Datalog Facts:

```
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-iterpreter, and a variable renaming $\sigma$, 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).





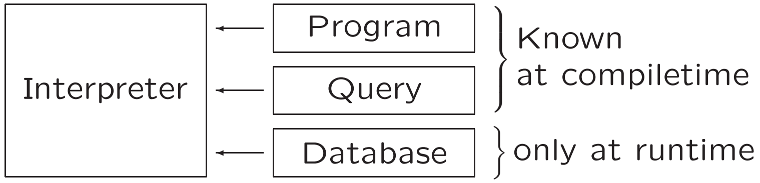
# Partial Evaluation (1)

  Interpreter
\+ Partial Evaluator

---

  Compiler

Inputs for the Meta-Interpreter:








# Partial Evaluation (2)

Idea:

- Fixpoint computation with conditional facts F ← 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 $X_0, X_1, \ldots$ 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:

$$path(X,Y) \leftarrow link(X,Y).$$
$$path(X,Z) \leftarrow link(X,Y) \wedge path(Y,Z).$$
$$?\ path(0,\ X).$$

Initial Set of Conditional Facts:

- `query(path(0,X))` $\longleftarrow$ `true.`
- `rule(path(X,Y),[link(X,Y)])` $\longleftarrow$ `true.`
- `rule(path(X,Z),[link(X,Y), path(Y,Z)])` $\longleftarrow$ `true.`
- `db(link(X,Y))` $\longleftarrow$ `link(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($(B_1, B_2)$, $(B'_1, B'_2)$): | $\sigma$ | |
| 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)     :- link(X,Y).
path(X,Z)     :- link(X,Y), path(Y,Z).
?- path(0, X).
```

Output:

```
p0(X0)        :- link(0,X0).
p1(X1)        :- link(0,X1).
p0(X0)        :- p1(X1), link(X1,X0).
p1(X1)        :- p1(X2), link(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.
  ```