Foundations of Data and Knowledge Systems

VU 181.212, WS 2010

6. Operational Semantics

Thomas Eiter and Reinhard Pichler

Institut für Informationssysteme
Technische Universität Wien

December 7, 2010

Outline

Evaluation Strategies

There are two basic evaluation strategies of rule bases:

1. *Forward Chaining*: In the spirit of *Modus Ponens*:

$$\frac{\varphi, \quad \varphi \Rightarrow \psi}{\psi}$$

   Apply the rules to conclude new facts (cf. immediate consequence operator $\mathbf{T}_S$).
   This leads to a *bottom-up* evaluation of rules, from the facts to the desired conclusion.

2. *Backward Chaining*: In the spirit of Abductive Reasoning:

$$\frac{\psi, \quad \varphi \Rightarrow \psi}{\varphi}$$

   Reduce proving ψ via a rule with consequent ψ to proving its antecedent φ.
   This leads to a *top-down* evaluation of rules, from a desired conclusion (goal) towards the facts.

Mixed forms of evaluation exist (realizing a bidirectional search).

Outline

Semi-Naive Evaluation

Recall

Datalog: a special case of Logic Programming

- No functions symbols, only constants; no negation
- Partitioning of the predicate symbols of a program P, called the schema of P, into
  - the set $ext(P)$ of extensional predicates, and
  - the set $int(P)$ of intensional predicates.

  Extensional predicates can not occur in rule heads. By default, all predicates occurring only in rule heads are assumed to be extensional.
- Usually, all variables in the consequent of a clause also occur in the antecedent (range-restriction, safety).

Semantically, a fact-free Datalog program P specifies a mapping from each Herbrand interpretation I of $ext(P)$ to one of $int(P)$ given by $HI(lfp(\mathbf{T}_{P \cup I_{|ext(P)}}))$.

($I_{|ext(P)}$ ... restriction of I to $ext(P)$).

Example

Program P (including extensional facts):

feeds_milk(betty).

lays_eggs(betty).

has_spines(betty).

monotreme(X)←
    lays_eggs(X), feeds_milk(X).

echidna(X)←
    monotreme(X), has_spines(X).

Example

Program P (including extensional facts):

feeds_milk(betty).

lays_eggs(betty).

has_spines(betty).

monotreme(X)←
    lays_eggs(X), feeds_milk(X).

echidna(X)←
    monotreme(X), has_spines(X).

Schema of P:

- $ext(P)$
- $int(P)$

Example

Program P (including extensional facts):

feeds_milk(betty).

lays_eggs(betty).

has_spines(betty).

monotreme(X)←

lays_eggs(X), feeds_milk(X).

echidna(X)←

monotreme(X), has_spines(X).

- Schema of P:
  { feeds_milk, lays_eggs, has_spines, monotreme, echidna }
- Calculation of $lfp(\mathbf{T}_P)$ ($b = betty$):

$$\mathbf{T}_P \uparrow 1 = \{b\}_{feeds}, \{b\}_{lays}, \{b\}_{spines}, \{\}_{monotreme}, \{\}_{echidna}$$
$$\mathbf{T}_P \uparrow 2 = \{b\}_{feeds}, \{b\}_{lays}, \{b\}_{spines}, \{b\}_{monotreme}, \{\}_{echidna}$$
$$\mathbf{T}_P \uparrow 3 = \{b\}_{feeds}, \{b\}_{lays}, \{b\}_{spines}, \{b\}_{monotreme}, \{b\}_{echidna}$$
$$= lfp(\mathbf{T}_P)$$

Naive Evaluation

Straight implementation of the immediate consequence operator $\mathbf{T}_P$:

```
I_0 := ∅
I_1 := ground_facts(P)
i := 1
while I_i ≠ I_{i-1} do
  i := i + 1
  I_i := I_{i-1}
  while (R = Rules.next())
    Insts := instantiations(R, I_{i-1})
    while (inst = Insts.next())
      I_i := I_i ∪ head(inst)
return I_i
```

instantiations (R, I): all instances r of rules in R s.t. body(r) is satisfied by I.

Disadvantage

Refiring of rules (e.g., all facts are reobtained in each step; monotreme(betty) again in Step 3).

Idea: only consider rules which involve *newly derived atoms*.

Semi-Naive Evaluation

incremental forward chaining:

```
KnownFacts := ∅
Ink := { Fact | (Fact ← true) ∈ P }
while (Ink ≠ ∅)
  Insts := instantiations(R, KnownFacts, Ink)
  KnownFacts := KnownFacts ∪ Ink
  Ink := heads(Insts)
return KnownFacts
```

instantiations (R, KnownFacts, Ink): all instances r of rules in R s.t. body(r) is satisfied by KnownFacts ∪ Ink using some fact from Ink.

- Further improvements: e.g.,
  - use only rule instances with head not in KnownFacts ∪ Ink
  - store partially instantiated rules (incremental satisfaction of the body)
  - in addition, share common body parts between rules ($\leadsto$ RETE Algorithm)
- Other view: map Datalog to Relational Algebra
  - Search solution for system of equations, using algebraic methods (e.g., Gauß-Seidel iteration (see [Ceri, Gottlob, Tanca 1990])
- Extensive treatment: [Abiteboul et al., 1995]

Outline

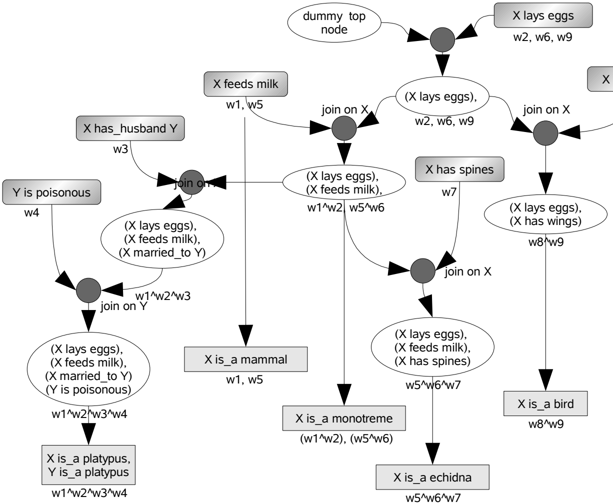
RETE Algorithm

- By Charles Forgy (1990), for forward chaining (production) systems
- Storage of partially instantiated rules
- Sharing of instantiated literals among similar rules
- Several optimizations, industrial use (Clips, Drools, JRules, ...)

Basic approach:

- Use
  - production memory PM (rule store) and
  - working memory WM (current facts)
- Different kinds of nodes:
  - alpha-node: represents a single atomic condition in rule bodies (across rules); it contains all WM elements that make it true;
  - beta-node: represents a conjunction of alpha-nodes; it contains tuples of WM elements satisfying them.
  - join-node: for computational purposes (combining alpha and/or beta nodes)
  - production-node: one per rule, holding all tuples of WM elements that satisfy its body.

Example



Working memory:
w1: anna feeds milk
w2: anna lays eggs
w3: anna married_to pierre
w4: pierre is poisonous
w5: betty feeds milk
w6: betty lays eggs
w7: betty has spines
w8: tux has wings
w9: tux lays eggs

Production memory:
p1: X lays eggs, X has wings ==> X is_a bird
p2: X feeds milk ==> X is_a mammal
p3: X feeds milk, X lays eggs ==> X is_a monotreme
p4: X feeds milk, X lays eggs, X has_spines ==> X is_a echidna
p5: X feeds milk, X lays eggs, X married_to Y, Y is poisonous ==> X is_a platypus, Y is_a platypus

Outline

SLD Resolution: Principles

- goal driven evaluation of logic programs (backward chaining)
- to show that $P \models \varphi$, show that $P \cup \{\neg\varphi\}$ is unsatisfiable
- uses unification and resolution: basically,

$$\frac{\varphi_1 \vee \psi, \quad \neg\psi \vee \varphi_2}{\varphi_1 \vee \varphi_2}$$

  (ψ ... atomic formula)
- recall that $\varphi \leftarrow \psi$ is equivalent to $\varphi \vee \neg\psi$
- SLD resolution: Selected Literal Definite Clause
- resolution with backtracking is used as control mechanism in Prolog

Observe

- A goal $\leftarrow a_1, \ldots, a_n$ is a syntactical variant of the first-order sentence $\forall x_1 \cdots \forall x_m (\bot \leftarrow a_1 \wedge \ldots \wedge a_n)$ where $x_1, \ldots, x_m$ are all variables occurring in $a_1, \ldots a_n$.
- This is equivalent to $\neg\exists x_1 \cdots \exists x_m (a_1 \wedge \ldots \wedge a_n)$.
- $P \models \exists x_1 \cdots \exists x_m (a_1 \wedge \ldots \wedge a_n)$ iff $P \cup \{\leftarrow a_1, \ldots, a_n\}$ is unsatisfiable

Definition (SLD Resolvent)

Let C be the clause $b \leftarrow b_1, \ldots, b_k$ and G a goal

$$\leftarrow a_1, \ldots, a_m, \ldots, a_n$$

such that G and C share no variables (otherwise, rename variables in C).

Then G' is an *SLD resolvent of G and C using ϑ*, if G' is the goal

$$\leftarrow (a_1, \ldots a_{m-1}, b_1, \ldots b_k, a_{m+1}, \ldots a_n)\vartheta$$

where ϑ is the mgu of a_m and b.

Definition (SLD Derivation)

An *SLD derivation* of $P \cup \{G\}$ consists of

- a sequence $G_0, G_1, \ldots$ of goals where $G = G_0$,
- a sequence $C_1, C_2, \ldots$ of variants of program clauses of P, and
- a sequence $\vartheta_1, \vartheta_2, \ldots$ of mgu's such that G_{i+1} is a resolvent from G_i and C_{i+1} using ϑ_{i+1}.

An *SLD-refutation* is a finite SLD-derivation whose last goal is empty.

Definition (SLD Tree)

An *SLD tree* T *w.r.t. a program* P *and a goal* G is a labeled tree where

- every node of T is a goal,
- the root of T is G, and
- if G is a node in T then G has a child G' connected to G by an edge labeled (C, ϑ) iff G' is an SLD-resolvent of G and C using ϑ.

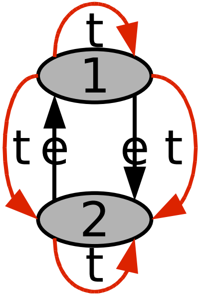
Definition (Computed Answer)

Given a definite program P and a definite goal G, a *computed answer* ϑ for $P \cup \{G\}$ is the substitution obtained by restricting the composition of the sequence of mgu's $\vartheta_1, \ldots \vartheta_n$ used in some SLD-refutation of $P \cup \{G\}$ to the variables occurring in G.

Example

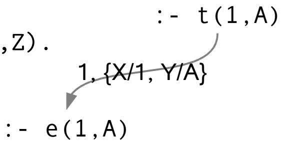
```
1: t(X,Y)←e(X,Y).
2: t(X,Z)←t(X,Y),e(Y,Z).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```

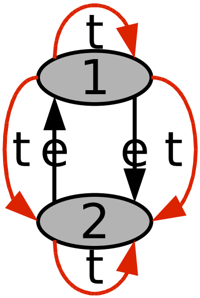
```
:- t(1,A)
```



Example

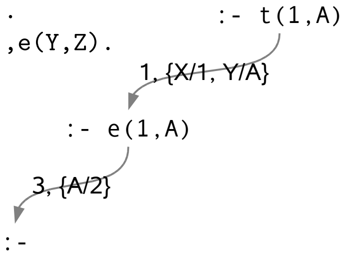
```
1: t(X,Y)←e(X,Y).
2: t(X,Z)←t(X,Y),e(Y,Z).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```

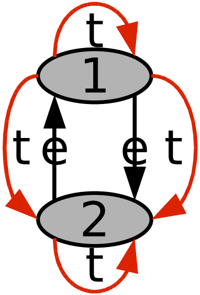




Example

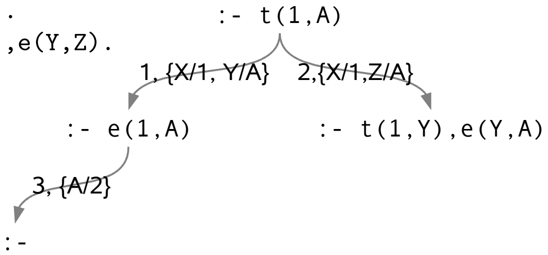
```
1: t(X,Y)←e(X,Y).
2: t(X,Z)←t(X,Y),e(Y,Z).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```

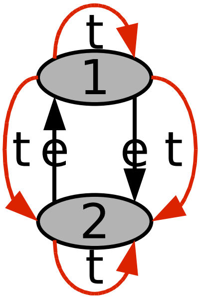




Example

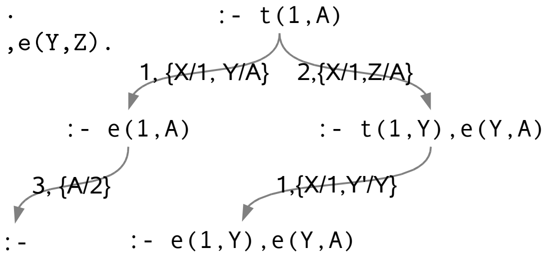
```
1: t(X,Y)←e(X,Y).
2: t(X,Z)←t(X,Y),e(Y,Z).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```

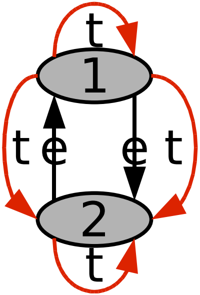




Example

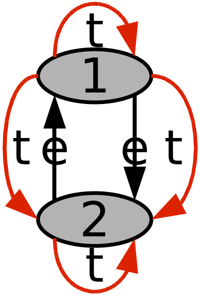
```
1: t(X,Y)←e(X,Y).
2: t(X,Z)←t(X,Y),e(Y,Z).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```

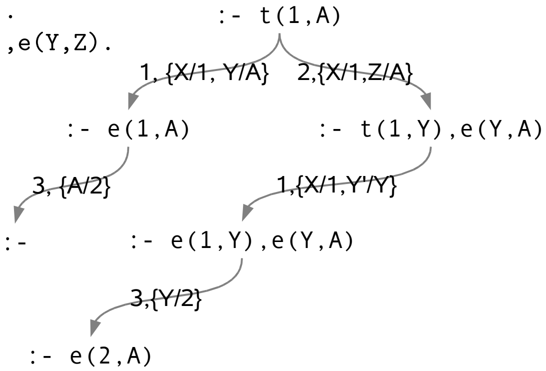




Example

```
1: t(X,Y)←e(X,Y).
2: t(X,Z)←t(X,Y),e(Y,Z).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```





Computation rules

In each resolution step, the selected literal and the clause C are chosen non-deterministically.

Definition (Computation Rule)

Call a function that maps to each goal one of its atoms a *computation rule*.

Proposition (Independence of the Computation Rule)

Let P be a definite program and G be a definite goal. Suppose there is an SLD-refutation of $P \cup \{G\}$ with computed answer ϑ. Then, for every computation rule R, there exists an SLD-refutation of $P \cup \{G\}$ using the atom selected by R as selected atom in each step with computed answer ϑ' such that $G\vartheta$ is a variant of $G\vartheta'$.

Let a *correct answer* for a program P and goal G be any substitution ϑ such that $P \models G\vartheta$.

Proposition (Soundness and Completeness of Logic Programming)

Let P be a program and let Q be a query. Then

- *every computed answer of P and G is a correct answer, and*
- *for every correct answer σ of P and G there exists a computed answer ϑ such that ϑ is more general that σ.*

Definition (SLD Procedure)

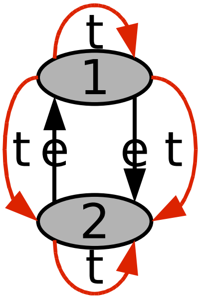
An *SLD-procedure* is any deterministic SLD-resolution algorithm constrained by

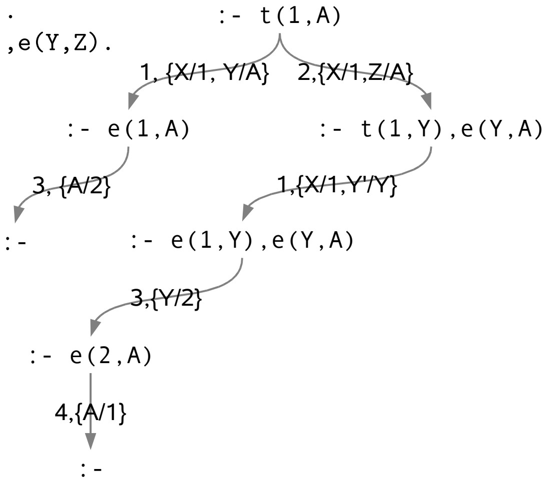
- a computation rule and
- an order for visiting the finite branches of an SLD-tree (*search strategy*).

- The completeness of a SLD procedure depends on the search strategy.
- To be complete, each leaf of a (finite) branch must be visited after finitely many steps (fairness).

Example (cont'd)

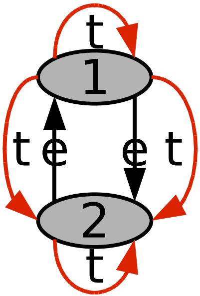
```
1: t(X,Y)←e(X,Y).
2: t(X,Z)←t(X,Y),e(Y,Z).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```

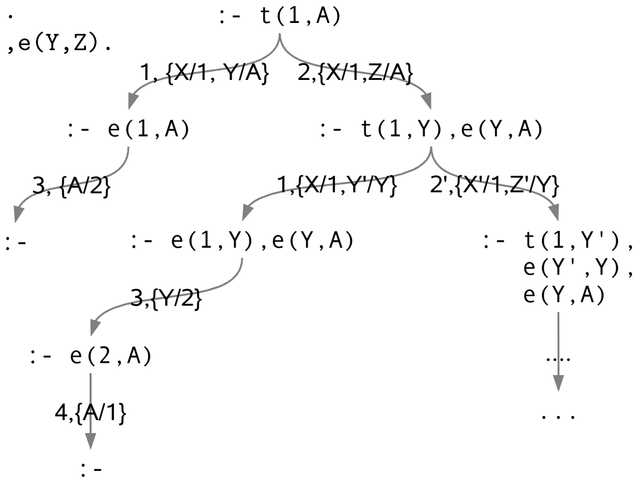




Example (cont'd)

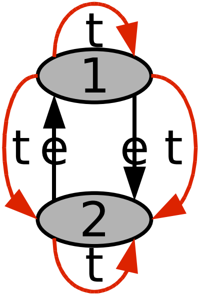
```
1: t(X,Y)←e(X,Y).
2: t(X,Z)←t(X,Y),e(Y,Z).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```

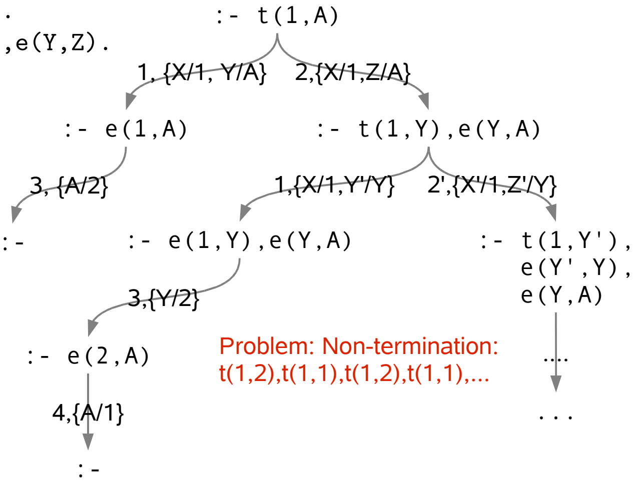




Example (cont'd)

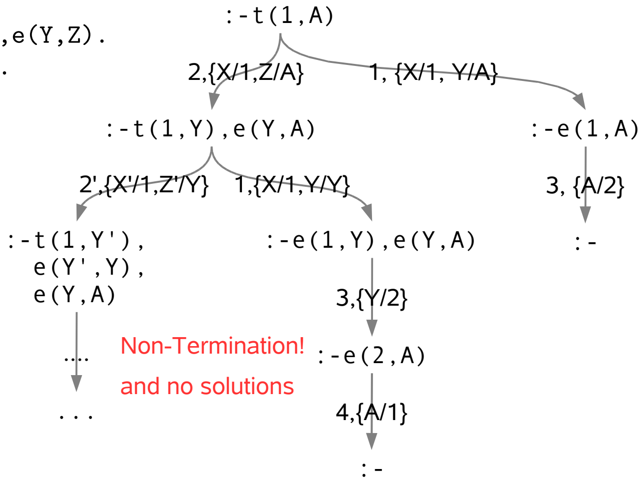
```
1: t(X,Y)←e(X,Y).
2: t(X,Z)←t(X,Y),e(Y,Z).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```





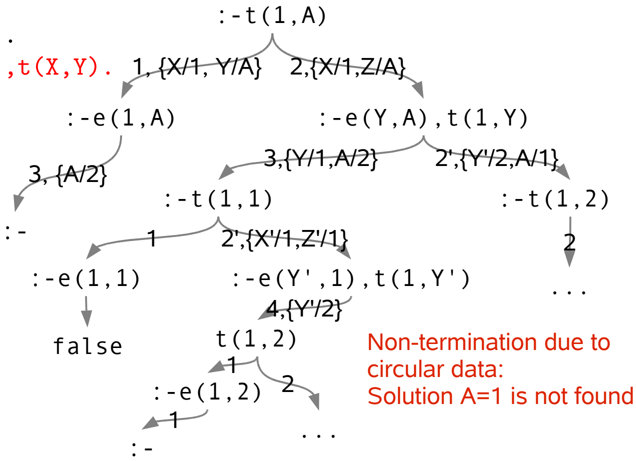
Example

```
2: t(X,Z)←t(X,Y),e(Y,Z).
1: t(X,Y)←e(X,Y).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```



Example

```
1: t(X,Y)←e(X,Y).
2: t(X,Z)←e(Y,Z),t(X,Y).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```



Outline

OLDT Resolution

- Non-termination of SLD resolution due to infinite branches
- Infinite branches $B = [G_0, G_1, \ldots]$ due to
  1. variants of the same goal on the infinite branch, i.e., in some subsequence $[G_{i_0}, G_{i_1}, \ldots]$, for all $j, k \in \mathbb{N}$, G_{i_j} and G_{i_k} contain an equal atom (up to renaming of variables) or
  2. subsuming goals on the infinite branch, i.e. in some subsequence $[G_{i_0}, G_{i_1}, \ldots]$, for all $j \in \mathbb{N}$, G_{i_j} contains an atom which is a real instance of an atom in $G_{i_{j-1}}$.

Ideas

- Avoid repeated evaluation of a subgoal on the same computation path through tabling or memorization, similar as in dynamic programming.
- Side effect: no repeated evaluations of subgoals at all.
- Use designated tabled predicates.
- Make distinction between solution nodes (goals) and lookup nodes (goals).

OLDT – Basic Elements

Definition (OLDT-structure)

An *OLDT-structure* (T, T_S, T_L) consists of

- an SLD-tree T,
- a *solution table* T_S, i.e., a set of pairs $(a, T_S(a))$ where
  - a is an atom and
  - $T_S(a)$ is a list of instances of a called the *solutions* of a, and
- a *lookup table* T_L, i.e., a set of pairs $(a, T_L(a))$ where a is an atom and $T_L(a)$ is a pointer to an element of $T_S(a')$ such that a is an instance of a'.

  T_L contains one pair $(a, T_L(a))$ for an atom a occurring as a leftmost atom of a goal in T.

- The initial OLDT-structure has as T the goal and void T_S and T_L.
- The OLDT-structure is stepwise extended, using SLD resolution and lookup, employing a left-to-right computation rule.

OLDT-Extension

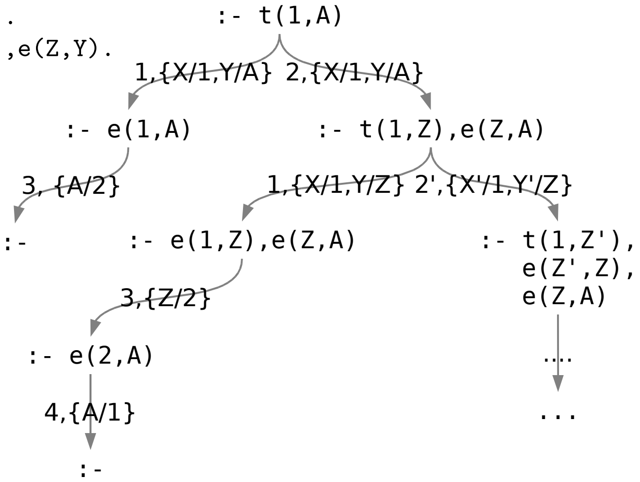
The *extension of an OLDT structure* (T, T_S, T_L) consists of three steps.

1. resolution step: a new goal G' is added to T, resolving some goal $G = \leftarrow a_1, \ldots, a_n$ in T that is
   (i) a non-tabled goal or a solution goal, with a clause C, resp.
   (ii) a lookup goal with the atom a from $T_L(a_1)$.
2. classification step: G' is
   - a non-table goal, if the leftmost atom of G', a'_1, has not a table predicate.
   - a table goal otherwise, and is
     - a lookup goal, if T_S has some $(a, T_S(a))$ where a is more general than a'_1. Then, add (a'_1, p) to T_L where p points to the first element of $T_S(a)$.
     - a solution node, if T_S contains no $(a, T_S(a))$ where a is more general than a'_1. In this case, add $(a'_1, [\,])$ to T_L.
3. table update step: add new solutions to T_S:
   - Suppose $G' = \leftarrow a_2, \ldots, a_n$ results from some table goal $G = \leftarrow a_1, \ldots, a_n$ in T by an SLD resolution $G_0, G_1, \ldots, G_m$ with $\vartheta_1, \vartheta_2, \ldots, \vartheta_m$.
   - Add the restriction ϑ of $\vartheta_1 \cdots \vartheta_m$ to the variables of a_1 as answer for a_1 to $T_S(a_1)$.

Example

1: t(X,Y)←e(X,Y).
2: t(X,Y)←t(X,Z),e(Z,Y).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).

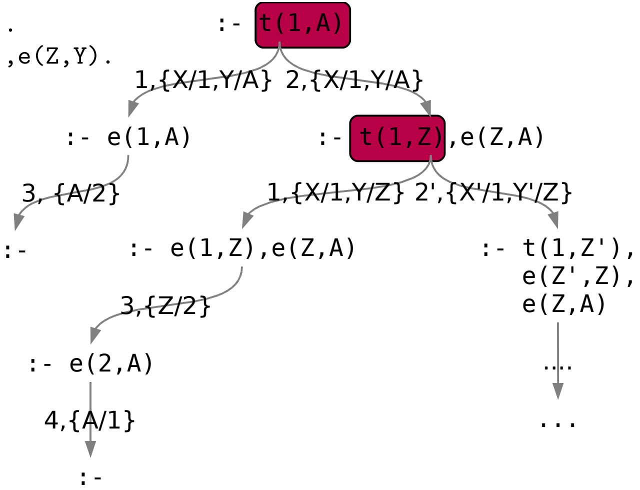
Let t be a table predicate



Example

```
1: t(X,Y)←e(X,Y).
2: t(X,Y)←t(X,Z),e(Z,Y).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```

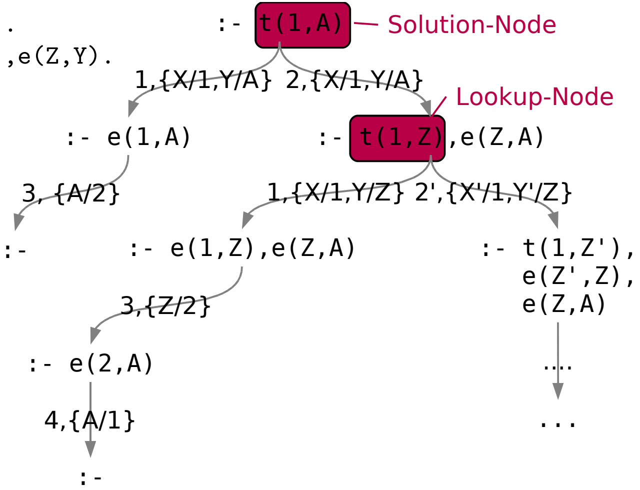
Let `t` be a table predicate



Example

1: t(X,Y)←e(X,Y).
2: t(X,Y)←t(X,Z),e(Z,Y).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).

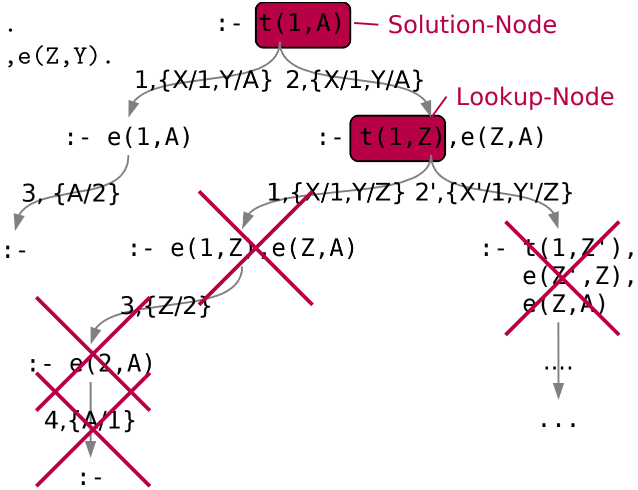
Let t be a table predicate



Example

```
1: t(X,Y)←e(X,Y).
2: t(X,Y)←t(X,Z),e(Z,Y).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).
```

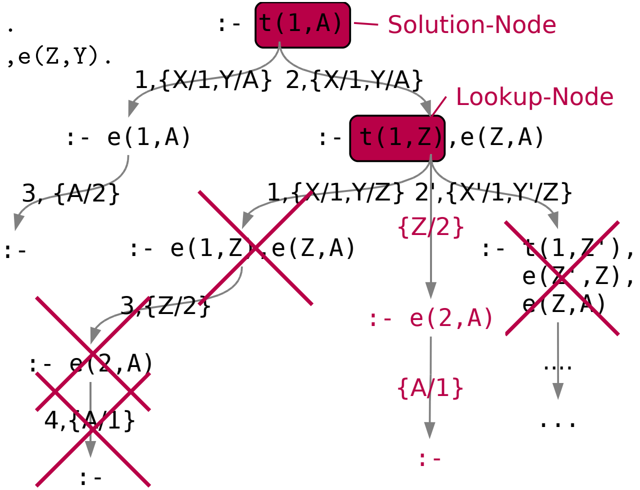
Let t be a table predicate



Example

1: t(X,Y)←e(X,Y).
2: t(X,Y)←t(X,Z),e(Z,Y).
3: e(1,2).
4: e(2,1).
5: ←t(1,A).

Let t be a table predicate



Completeness

OLDT-resolution is not complete in general

```
p(x) ← q(x), r .
q(s(x)) ← q(x) .
q(a) ← .
r ← .
← p(x) .
```

Problem

- Reduction steps are only applied to lookup goal $\leftarrow q(x'), r$.
- No solutions for $p(x)$ will be produced in finite time.

Remedy

Special search strategy (multi-stage depth first, MSDFS): Order the nodes in the OLDT tree, avoid repeating reduction of a node if other nodes are available.

Above: avoid reducing the lookup goal $\leftarrow q(x'), r$. twice, and reduce $\leftarrow r$.

- Under MSDFS, OLDT-resolution becomes complete.

Outline

Magic Templates Transformation

Until now, we have seen:

- forward chaining (data driven) evaluation of LP
- backward chaining (goal driven) evaluation of LP
- improvement of backward chaining by tabling

Idea of the magic templates transformation:

- take the best of both worlds:
  - Efficiency of goal directedness
  - Good termination properties of forward chaining
  - Easy implementation of a forward chaining rule engine

Example

```
t(X,Y) ← r(X,Y)
t(X,Z) ← t(X,Y), t(Y,Z)
r(a,b).
r(b,c).
r(c,d).
```

Example

```
t(X,Y) ← r(X,Y)
t(X,Z) ← t(X,Y), t(Y,Z)
r(a,b).
r(b,c).
r(c,d).
← t(b, Answer).
```

Goal-directed evaluation

- Bottom-up evaluation produces many facts:
  t(a,b), t(a,c), t(a,d), t(b,c), t(b,d), t(c,d)
- Only t(b,c), t(b,d) are relevant for query answers.

Example

```
t(X,Y) ← r(X,Y)
r(a,b).
r(b,c).
r(c,d).
← t(b, Answer).
```

Goal-directed evaluation

- Bottom-up evaluation produces many facts:
  t(a,b), t(a,c), t(a,d), t(b,c), t(b,d), t(c,d)
- Only t(b,c), t(b,d) are relevant for query answers.
- Idea:
  - Utilize information about which variables in atom are bound or free for evaluation.
  - Rewrite the program into an adorned program, respecting binding patterns.
  - Transform the adorned program into a set of rules that can be efficiently evaluated bottom up.

Adornment of Datalog programs

Sideways Information Passing Strategy

A sideways information passing strategy (SIPS) determines how variable bindings gained from the unification of a rule head with a goal or sub-goal are passed to the body of the rule, and how they are passed from a set of literals in the body to another literal.

- Evaluation in Prolog implements a special SIPS (head-to-body, left to right).
- Many other SIPS might be convenient.
- W.l.o.g., the query Q is of form $\leftarrow q(t_1, \ldots, t_n)$.

Binding Pattern

A binding pattern for an n-ary predicate is a string $x_1 \cdots x_n$, $n \geq 0$, where each $x_i \in \{b, f\}$ (intuitively, b means "bound" and f means "free").
The binding pattern for the query atom $q(t_1, \ldots, t_n)$ is $x_1 \cdots x_n$ such that $x_i = b$ iff t_i is a constant.

Rule Adornment

- Given a rule

  $$p(t_1, \ldots, t_n) \leftarrow p_1(t_{1,1}, \ldots, t_{1,n_1}), \ldots p_m(t_{m,1}, \ldots, t_{m,n_m})$$

  and a binding pattern $bp = x_1 \cdots x_n$ for p, the rule adorned with bp,

  $$p^{bp}(t_1, \ldots, t_n) \leftarrow p_1^{a_1}(t_{1,1}, \ldots, t_{1,n_1}), \ldots p_m^{a_m}(t_{m,1}, \ldots, t_{m,n_m}),$$

  is constructed left to right, where for extensional p_i, $a_i = \epsilon$ and otherwise in $a_i = x_{i,i_1} \cdots x_{i,n_i}$ we have $x_{i,j} = b$ iff $t_{i,j}$ is either a constant or equal some t'_j or some $t_{i',j'}$ where $i' < i$.
- Starting with the binding pattern bp for the query atom $q(t_1, \ldots, t_n)$, all rules whose head unifies with the query atom $q(t_1, \ldots, t_n)$ are adorned with bp.
- Recursively, for each adorned atom $p_i^{a_i}(t_{i,1}, \ldots, t_{i,n_i})$, all rules whose head unifies with $p_i(t_{i,1}, \ldots, t_{i,n_i})$ are adorned with a_i.

Example

Goal-directed evaluation

```
t(X,Y) ← r(X,Y)
t(X,Z) ← t(X,Y), t(Y,Z)
r(a,b).
r(b,c).
r(c,d).
← t(b, Answer).
```

Example

$\mathtt{t(X,Y)} \leftarrow \mathtt{r(X,Y)}$
$\mathtt{t_1(X,Z)} \leftarrow \mathtt{t_2(X,Y),\ t_3(Y,Z)}$
r(a,b).
r(b,c).
r(c,d).
$\leftarrow$ t(b, Answer).

Goal-directed evaluation

Label occurrences of t for better distinction

Information passing

- t $\hookrightarrow_X$ r.
- $\mathtt{t_1} \hookrightarrow_X \mathtt{t_2}$.
- $\mathtt{t_2} \hookrightarrow_Y \mathtt{t_3}$.

Example

```
t^bf(X,Y) ← r(X,Y)
t1^bf(X,Z) ← t2^bf(X,Y), t3^bf(Y,Z)
r(a,b).
r(b,c).
r(c,d).
← t(b, Answer).
```

Goal-directed evaluation

Label occurrences of t for better distinction

Information passing

- t $\hookrightarrow_X$ r.
- t_1 $\hookrightarrow_X$ t_2.
- t_2 $\hookrightarrow_Y$ t_3.

Adornment (**b**ound, **f**ree)

Goal-Directed Rewriting

- Given the adorned program P^{ad}, transform it into a program P_m^{ad} such that all sub-goals relevant for answering Q can be computed from additional rules in P_m^{ad}.
- Intuition: provide possible values for the bound arguments of a predicate (*magic sets*).

Method:

1. For each adorned predicate $\texttt{p}^a$, create a predicate $\texttt{magic\_p}^a$ whose arity is the number of b's in a.
2. For the query atom $q(t_1, \ldots, t_n)$ with binding pattern a, add to P^{ad} a fact $\texttt{magic\_q}^a(c_1, \ldots, c_m)$ where $c_1, \ldots, c_m$ are the constants among $t_1, \ldots, t_n$ (seed).
3. Introduce rules for computing subgoals reflecting SIP.

   For
   $$p^{bp}(t_1, \ldots, t_n) \leftarrow p_1^{a_1}(\vec{t_1}), \ldots, p_m^{a_m}(\vec{t_m}) \quad (1)$$
   add to P^{ad} for $j \leq j < m$ rules
   $$\texttt{magic\_p}_{j+1}^{a_{j+1}}(x_1, \ldots, x_{n_{j+1}}) \leftarrow \texttt{magic\_p}^{bp}(t_1, \ldots, t_n), p_1(\vec{t_1}), \ldots, p_j(\vec{t_j}) \quad (2)$$
   where p_{j+i} is intensional and $x_1, \ldots, x_{n_{j+1}}$ are the bound variables among $\vec{t}_{j+1}$.
4. Adapt the original rules (1) of P^{ad}.

   Add in the body
   - $\texttt{magic\_p}^{bp}(t_1, \ldots, t_n)$,
   - $\texttt{magic\_p}_{j+1}^{a_{j+1}}(x_1, \ldots, x_{n_{j+1}})$ for each magic rule (2) above, unless all x_i are bound by extensional predicates.

Example

$\texttt{t}^{bf}\texttt{(X,Y)} \leftarrow \texttt{r(X,Y)}$
$\texttt{t}_1{}^{bf}\texttt{(X,Z)} \leftarrow \texttt{t}_2{}^{bf}\texttt{(X,Y)},\ \texttt{t}_3{}^{bf}\texttt{(Y,Z)}$
r(a,b).
r(b,c).
r(c,d).
$\leftarrow$ t(b, Answer).

Goal-directed evaluation
Information passing

- $\texttt{t} \hookrightarrow_X \texttt{r}$.
- $\texttt{t}_1 \hookrightarrow_X \texttt{t}_2$.
- $\texttt{t}_2 \hookrightarrow_Y \texttt{t}_3$.

Adornment (**b**ound, **f**ree)

Example

$\mathtt{t}^{bf}\mathtt{(X,Y)} \leftarrow \mathtt{r(X,Y)}$
$\mathtt{t}_1{}^{bf}\mathtt{(X,Z)} \leftarrow \mathtt{t}_2{}^{bf}\mathtt{(X,Y)},\ \mathtt{t}_3{}^{bf}\mathtt{(Y,Z)}$
r(a,b).
r(b,c).
r(c,d).
$\leftarrow$ t(b, Answer).

Goal-directed evaluation

Information passing

- $\mathtt{t} \hookrightarrow_X \mathtt{r}$.
- $\mathtt{t}_1 \hookrightarrow_X \mathtt{t}_2$.
- $\mathtt{t}_2 \hookrightarrow_Y \mathtt{t}_3$.

Adornment (**b**ound, **f**ree)

$\mathtt{magic\_t}^{bf}\mathtt{(b)}$.

seed

Example

$\mathtt{t}^{bf}\mathtt{(X,Y)} \leftarrow \mathtt{r(X,Y)}$
$\mathtt{t_1}^{bf}\mathtt{(X,Z)} \leftarrow \mathtt{t_2}^{bf}\mathtt{(X,Y)},\ \mathtt{t_3}^{bf}\mathtt{(Y,Z)}$
r(a,b).
r(b,c).
r(c,d).
$\leftarrow$ t(b, Answer).

Goal-directed evaluation
Information passing

- $\mathtt{t} \hookrightarrow_X \mathtt{r}$.
- $\mathtt{t_1} \hookrightarrow_X \mathtt{t_2}$.
- $\mathtt{t_2} \hookrightarrow_Y \mathtt{t_3}$.

Adornment (**b**ound, **f**ree)

$\mathtt{magic\_t}^{bf}\mathtt{(b)}.$
$\mathtt{magic\_t}^{bf}\mathtt{(X)} \leftarrow \mathtt{magic\_t}^{bf}\mathtt{(X)}.$

seed
Magic Rules

Example

t^{bf}(X,Y) ← r(X,Y)
t_1^{bf}(X,Z) ← t_2^{bf}(X,Y), t_3^{bf}(Y,Z)
r(a,b).
r(b,c).
r(c,d).
← t(b, Answer).

Goal-directed evaluation
Information passing

- t $\hookrightarrow_X$ r.
- t_1 $\hookrightarrow_X$ t_2.
- t_2 $\hookrightarrow_Y$ t_3.

Adornment (**b**ound, **f**ree)

---

magic_t^{bf}(b).
magic_t^{bf}(X) ← magic_t^{bf}(X).
magic_t^{bf}(Y) ← magic_t^{bf}(X), t(X,Y).

seed
Magic Rules

Example

$t^{bf}(X,Y) \leftarrow r(X,Y)$
$t_1{}^{bf}(X,Z) \leftarrow t_2{}^{bf}(X,Y),\ t_3{}^{bf}(Y,Z)$
r(a,b).
r(b,c).
r(c,d).
$\leftarrow$ t(b, Answer).

Goal-directed evaluation

Information passing

- $t \hookrightarrow_X r$.
- $t_1 \hookrightarrow_X t_2$.
- $t_2 \hookrightarrow_Y t_3$.

Adornment (**b**ound, **f**ree)

$magic\_t^{bf}(b)$. — seed

$magic\_t^{bf}(X) \leftarrow magic\_t^{bf}(X)$.
$magic\_t^{bf}(Y) \leftarrow magic\_t^{bf}(X),\ t(X,Y)$. — Magic Rules

$t(X,Y) \leftarrow magic\_t^{bf}(X),\ r(X,Y)$.

Rewritten Rules

Example

t^{bf}(X,Y) ← r(X,Y)
$t_1{}^{bf}$(X,Z) ← $t_2{}^{bf}$(X,Y), $t_3{}^{bf}$(Y,Z)
r(a,b).
r(b,c).
r(c,d).
← t(b, Answer).

Goal-directed evaluation
Information passing

- t $\hookrightarrow_X$ r.
- t_1 $\hookrightarrow_X$ t_2.
- t_2 $\hookrightarrow_Y$ t_3.

Adornment (**b**ound, **f**ree)

---

magic_t^{bf}(b).
magic_t^{bf}(X) ← magic_t^{bf}(X).
magic_t^{bf}(Y) ← magic_t^{bf}(X), t(X,Y).
t(X,Y) ← magic_t^{bf}(X), r(X,Y).
t(X,Z) ← magic_t^{bf}(X),t(X,Y),
magic_t^{bf}(Y),t(Y,Z).

seed
Magic Rules

Rewritten Rules

Example

t^{bf}(X,Y) ← r(X,Y)
$t_1{}^{bf}$(X,Z) ← $t_2{}^{bf}$(X,Y), $t_3{}^{bf}$(Y,Z)
r(a,b).
r(b,c).
r(c,d).
← t(b, Answer).

Goal-directed evaluation

Information passing

- t $\hookrightarrow_X$ r.
- t_1 $\hookrightarrow_X$ t_2.
- t_2 $\hookrightarrow_Y$ t_3.

Adornment (**b**ound, **f**ree)

---

magic_t^{bf}(b).
magic_t^{bf}(X) ← magic_t^{bf}(X).
magic_t^{bf}(Y) ← magic_t^{bf}(X), t(X,Y).
t(X,Y) ← magic_t^{bf}(X), r(X,Y).
t(X,Z) ← magic_t^{bf}(X),t(X,Y),
         magic_t^{bf}(Y),t(Y,Z).
r(a,b). r(b,c). r(c,d).

seed

Magic Rules

Rewritten Rules

Extensional Facts

Example

t^{bf}(X,Y) ← r(X,Y)
t_1^{bf}(X,Z) ← t_2^{bf}(X,Y), t_3^{bf}(Y,Z)
r(a,b).
r(b,c).
r(c,d).
← t(b, Answer).

Goal-directed evaluation
Information passing

- t $\hookrightarrow_X$ r.
- t_1 $\hookrightarrow_X$ t_2.
- t_2 $\hookrightarrow_Y$ t_3.

Adornment (**b**ound, **f**ree)

---

magic_t^{bf}(b).
magic_t^{bf}(X) ← magic_t^{bf}(X).
magic_t^{bf}(Y) ← magic_t^{bf}(X), t(X,Y).
t(X,Y) ← magic_t^{bf}(X), r(X,Y).
t(X,Z) ← magic_t^{bf}(X),t(X,Y),
magic_t^{bf}(Y),t(Y,Z).
r(a,b). r(b,c). r(c,d).

Evaluation:
magic_t^{bf}(b).
t(b,c).
magic_t^{bf}(c).
t(c,d).
magic_t^{bf}(d).
t(b,d).

Magic Set Transformation with Negation

Problem with negation

- Even for stratified programs, the magic set transformation (MST) may have unstratified outcome.

Causes for unstratification of the MST

1. positive and negative occurrence of a literal in a rule body
2. multiple negative occurrences of a literal in a rule body
3. negative literal in a recursive rule

Solution (Source 1)

- distinction of contexts of problematic atoms:
  label occurrences of predicate p to p_1, p_2 etc
- replicate each rule defining p with p_i (also in the body), for all p_i

Magic Set Transformation with Negation

Problem with negation

- Even for stratified programs, the magic set transformation (MST) may have unstratified outcome.

Causes for unstratification of the MST

1. positive and negative occurrence of a literal in a rule body
2. multiple negative occurrences of a literal in a rule body
3. negative literal in a recursive rule

Solution (Source 1)

- distinction of contexts of problematic atoms: label occurrences of predicate p to p_1, p_2 etc
- replicate each rule defining p with p_i (also in the body), for all p_i

Magic Set Transformation with Negation

Problem with negation

- Even for stratified programs, the magic set transformation (MST) may have unstratified outcome.

Causes for unstratification of the MST

1. positive and negative occurrence of a literal in a rule body
2. multiple negative occurrences of a literal in a rule body
3. negative literal in a recursive rule

Solution (Source 1)

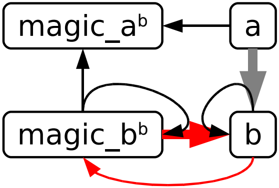
- distinction of contexts of problematic atoms:
  label occurrences of predicate p to p_1, p_2 etc
- replicate each rule defining p with p_i (also in the body), for all p_i

Example

```
a(x) ← not b(x), c(x,y), b(y).
b(x) ← c(x,y), b(y).
```

$\texttt{magic\_a}^b\texttt{(1).}$
$\texttt{magic\_b}^b\texttt{(x)} \leftarrow \texttt{magic\_a}^b\texttt{(x)}$
$\texttt{magic\_b}^b\texttt{(y)} \leftarrow$
  $\texttt{magic\_a}^b\texttt{(x), not b(x), c(x,y).}$
$\texttt{a(x)} \leftarrow$
  $\texttt{magic\_a}^b\texttt{(x), not b(x), c(x,y), b(y).}$
$\texttt{magic\_b}^b\texttt{(y)} \leftarrow \texttt{magic\_b}^b\texttt{(x), c(x,y).}$
$\texttt{b(x)} \leftarrow \texttt{magic\_b}^b\texttt{(x), c(x,y), b(y).}$

- b occurs both negatively and positively in the first rule.

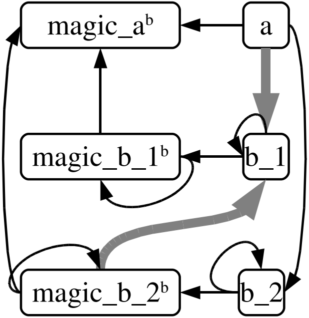


- Resulting program unstratifiable!

```
a(x) ← not b_1(x), c(x,y), b_2(y).
b_1(x) ← c(x,y), b_1(y).
b_2(x) ← c(x,y), b_2(y).
```

```
magic_a^b(1).
magic_b_1^b(x) ← magic_a^b(x).
magic_b_2^b(y) ←
   magic_a^b(x), not b_1(x), c(x,y).
magic_b^b(y) ←
   magic_a^b(x), not b(x), c(x,y).
a(x) ←
   magic_a^b(x), not b(x), c(x,y), b(y).
magic_b_i^b(y) ← magic_b_i^b(x), c(x,y).
b_i(x) ← magic_b_i^b(x), c(x,y), b_i(y).
i=1,2
```

- Context labeling of predicates
- Rule replication



- Result is stratifiable!

The second and third source of unstratifiability can be eliminated on the adorned rule set (preprocessing).

Magic Sets for Unstratified Programs

Also for unstratified logic programs under stable model semantics, MST can be developed.

E.g., [Faber et al., ICDT 2005/JCSS 2007]:

- Geared towards query answering, assuming that the program has some stable model.
- They introduced a suitable notion of *module* and *independent set*, to focus computation on a subprogram.
- The method makes also body-to-head propagation of values.
- fruitful application of magic sets e.g. in the area of data integration (INFOMIX project).

Outline

Well-Founded Semantics

Recall

- Idea: leave truth value incase of cyclic negation open (e.g., $p \leftarrow \neg p$)
- Use three-valued interpretations I (true, false, *undefined*), viewed as sets of ground literals.
- Employ *unfounded sets* to make atoms definitely false; a unique maximal (=greatest) unfounded set exists for any interpretation I.
- Define monotonic operators $\mathbf{T}_S(I)$ (immediate consequences), $\mathbf{U}_S$ (greatest unfounded set), $\mathbf{W}_S = \mathbf{T}_S \cup \mathbf{U}_S$
- The *well-founded model* of a set of normal clauses S is given by $lfp(\mathbf{W}_S)$; it may be partial or total

Problem

Computing unfounded set $\mathbf{U}_S$ (guessing)

A possible solution: *Alternating Fixpoint Procedure*

The Alternating Fixpoint Procedure

Central Idea

- Iteratively build up a set of negative conclusions $\tilde{A}$, which *underestimates* the set of atoms that are false in WFS.
- The derivation of positive conclusions from the eventual $\tilde{A}$ straightforward.

Method:

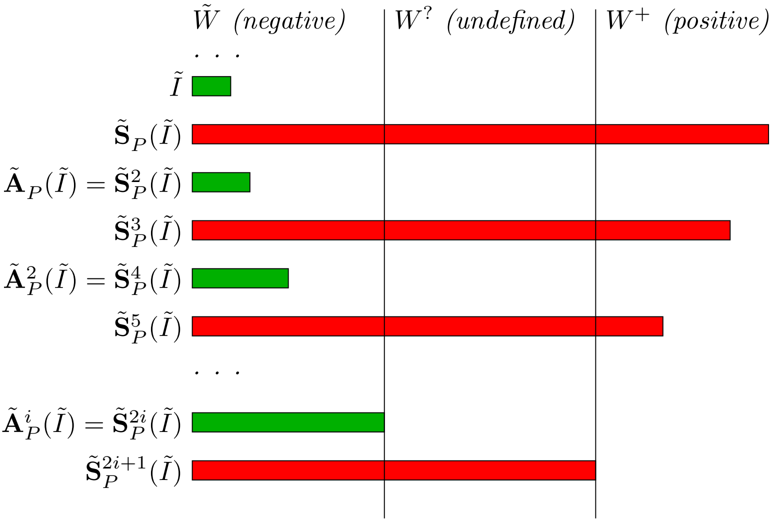
- Each iteration is a two-phase process
- Suppose $\tilde{I}$ is an *underestimate* of the negative conclusions under WFS
  1. Transform $\tilde{I}$ into an *overestimate* by

     $$\tilde{\mathbf{S}}_P(\tilde{I}) := \overline{lfp(\mathbf{T}_{P_{\tilde{I}}})} := \neg \cdot (HB_P - lfp(\mathbf{T}_{P_{\tilde{I}}})),$$

     where $P_{\tilde{I}} = P \cup \tilde{I}$, viewing negated predicates as new predicate symbols (HB_P ... Herbrand base of P)
  2. Transform the *overestimate* back to an *underestimate* by

     $$\mathbf{A}_P(\tilde{I}) := \tilde{\mathbf{S}}_P(\tilde{\mathbf{S}}_P(\tilde{I}))$$
- We have $\tilde{I} \subseteq \mathbf{A}_P(\tilde{I}) = \tilde{\mathbf{S}}^2_P(\tilde{I})$; initially, set $\tilde{I} = \emptyset$.

Alternating Fixpoint Procedure



Alternating Fixpoint Procedure: Example

$a \leftarrow c, \neg b.$
$b \leftarrow \neg a.$
$c.$

$p \leftarrow q, \neg s.$
$p \leftarrow r, \neg s.$
$p \leftarrow t.$
$q \leftarrow p.$
$r \leftarrow q.$
$r \leftarrow \neg c.$

- $HB_P = \{a, b, c, p, q, r, s, t\}$
- $\tilde{I}_0 = \emptyset$
- $lfp(\mathbf{T}_{P \cup \tilde{I}_0}) = \{c\}$
- $\tilde{I}_1 = \tilde{\mathbf{S}}_P(\tilde{I}_0) = \neg \cdot (HB_P - lfp(\mathbf{T}_{P \cup \tilde{I}_0})) = \{\neg a, \neg b, \neg p, \neg q, \neg r, \neg s, \neg t\}$
- $lfp(\mathbf{T}_{P \cup \tilde{I}_1}) = \{c, a, b\}$
- $\tilde{I}_2 = \tilde{\mathbf{S}}_P(\tilde{I}_1) = \neg \cdot (HB_P - lfp(\mathbf{T}_{P \cup \tilde{I}_1})) = \{\neg p, \neg q, \neg r, \neg s, \neg t\}$
- $\tilde{I}_3 = \tilde{I}_1$ and $\tilde{I}_4 = \tilde{I}_2$. Fixpoint reached!
- The well-founded model is $\{c, \neg p, \neg q, \neg r, \neg s, \neg t\}$.

Note

- For propositional program P, the AFP computation is polynomial.
- It is unknown whether for such P, the well-founded model is computable in linear time.