Query languages (NDBI049)

Datalog

Jaroslav Pokorný

MFF UK, Praha

jaroslav.pokorny@matfyz.cuni.cz

Terminology and constraints

- terms: variables or constants
- facts are atomic formulas containing only constants
- rules are Horn clauses

  L_0:- $L_1,\ldots,L_n$

  where L_i are atomic (positive) formulas
- atomic formulas or negations of atomic formulas are called literals.
- positive and negative literals
- facts are called basic literals

Terminology and constraints

- structure of rules:

  L_0 head of a rule

  $L_1,\ldots,L_n$ body of a rule

Remark: Facts and literals are also Horn clauses.

DATALOG - syntax and semantics (1)

1. Datalog program is a collection of facts and rules.
2. Three kinds of predicate symbols:
   - $R_i \in \mathbf{R}$
   - S_i ... virtual relations
   - built-in predicates $\leq, \geq, \neq, <, >, =$

   R_i and S_i are called ordinary.

   Remark: $\neq$ will not conceived as a negation (we will compare only bound variables)
3. Semantics of logic programs can be built by at least in three different ways:
   - proof theoretic,
   - model theoretic,
   - with fixpoints.

DATALOG - syntax and semantics (2)

- proof theoretic approach

Method: interpretation of rules as axioms usable to a proof, i.e. we make substitutions in body of rules and derive new facts from heads of rules. In the case of Datalog, it is possible to obtain *just all* derivable facts.

- model theoretic approach

Method: to predicate symbols we associate relations (a logical model) which satisfy the rules.

Ex.: Consider a logical program LP

IDB: P(x) :- Q(x)
Q(x) :- R(x),

i.e. Q and P denote virtual relations.

DATALOG - syntax and semantics (3)

- Let: R(1) Q(1) P(1)
  Q(2) P(2) M_1
  P(3)

Relations P*, Q*, R* make a model M_1 of the logical program LP.

- Let: R(1) (and other facts have value FALSE). Then relations P*, Q*, R* are not a model of the LP.
- Let: R(1) Q(1) P(1) M_2

then relations P*, Q*, R* make a model M_2 of the LP.

Let EDB: R(1), i.e. relational DB is given as
$R^* = \{(1)\}$.

then M_1 and M_2 are with the given DB consistent.

DATALOG - syntax and semantics (4)

- M_2 is even a minimal model, i.e. when we change anything there, we destroy consistency.
- M_1 does not make a minimal model.

Remark: with both semantics we obtain the same result.

Disadvantages of both approaches: non-effective algorithms in the case, when EDB is given by database relations.

DATALOG - dependency graph (1)

- with fixpoints

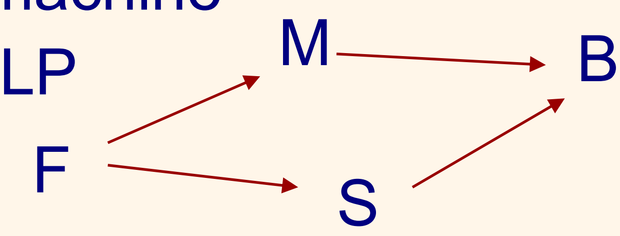
Method: evaluating algorithm+relational DB machine

Df.: dependency graph of a logical program LP

nodes: predicates from **R** and IDB

edges: (U,V) is an edge, if there is a rule

in :- … U ...



Ex.: extension of the original example

M(x):- F(x,y)

S'(y,w) :- F(x,y), F(x,w), $y \neq w$

B(x,y) :- S'(x,y), M(x)

C(x,y) :- F(x_1,x), F(x_2,y), S'(x_1,x_2)

C(x,y) :- F(x_1,x), F(x_2,y), C(x_1,x_2)

DATALOG - dependency graph (2)

R(x,y) :- S‘(x,y)

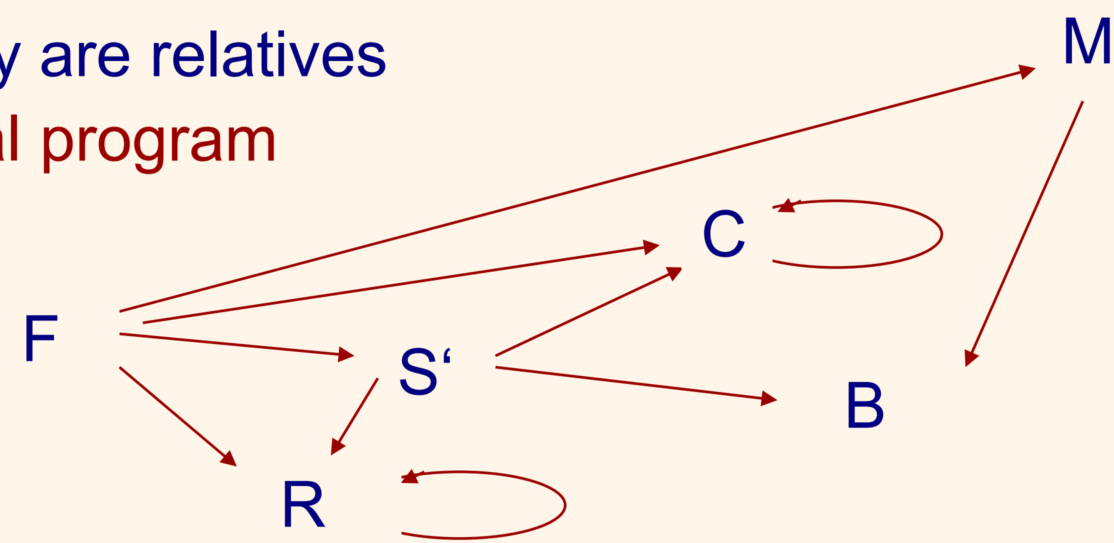
R(x,y) :- R(x,z), F(z,y)

R(x,y) :- R(z,y), F(z,x)

where C(x,y) … x is a cousin of y, i.e. their fathers are brothers

R(x,y) … x and y are relatives

recursive datalogical program



DATALOG - dependency graph (3)

R, C … recursive predicates

Df.: A logical program is recursive if there is a cycle in its dependency graph.

DATALOG - safe rules

Df.: safe rule

A variable x occurring in a rule is limited, if it occurs in the body of literal L of the same rule, where:

- L is given by an ordinary predicate, or
- L is of form x = a or a = x, or
- L is of form x=y or y=x and y is limited.

A rule is safe, if all its variables are limited.

Ex.: safety of rules

IS_GREATER_THAN(x,y) :- $x > y$

FRIENDS(x,y) :- M(x)

S‘(y,w) :- F(x,y), F(x,w), $y \neq w$

Is safe

Non-recursive DATALOG

- Its dependency graph is acyclic.
- There is a topological ordering of nodes such, that $R_i \rightarrow R_j$ implies $i < j$.

Remark: ordering is not given unambiguously

Ex.: ordering F - M - S - B

Non-recursive DATALOG

Principle of the algorithm (for one virtual relation):

(1) $U(x_1,\ldots,x_k)$:- $V_1(x_{i1},\ldots,x_{ik}),\ldots,\ V_s(x_{j1},\ldots,x_{js})$

transform to joins and selection

(2) for U it is performed

apply a projection on the result

(3) Steps (1), (2) are performed for all rules with U in their heads and for partial results

apply a union

Remark: Due to the acyclicity and topological ordering, the steps (1), (2) can be always applied for a rule.

Non-recursive DATALOG

Convention: variable x $\rightarrow$ attribute X

Rule rewriting:

- C(x,y) :- F(x_1,x), F(x_2,y), S'(x_1,x_2)

  1. step:

  AUX(X1,X,X2,Y) = F(X1,X) * F(X2,Y) * S'(X1,X2)

  2. step:

  C(X,Y) = AUX[X,Y]

- for S'

  S'(Y,W) = (F(X,Y) * F(X,W)) (Y$\neq$ W)[Y,W]

Non-recursive DATALOG

Other possibilities:

- V(x,y) :- P(a,x), R(x,x,z), U(y,z)

  1. and 2. step:

  V(.,.) = (P(1=a)[2] * R(1=2)[1,3] * U)[.,.]

Problem: In the rule head, constants, the same variables, and different orders of variables can occur.

A request on a rectification, i.e., a transformation of rules in such way, that heads with the same predicate symbol have a tuple of the same variables.

Non-recursive DATALOG

Ex.: P(a,x,y) :- R(x,y)

P(x,y,x) :- R(y,x)

We introduce u, v, w and do the substitutions:

P(u,v,w) :- R(x,y), u = a, in = x, w = y

P(u,v,w) :- R(y,x), u = x, in = y, w = x

$\Rightarrow$ P(u,v,w) :- R(v,w), u = a,

P(u,v,w) :- R(v,u), w = u

Lemma:

(1) If the rule is safe, then after rectification too.

(2) The original and rectified rule are equivalent, i.e., after its evaluation we obtain the same relation.

Non-recursive DATALOG

Statement: The evaluated program provides for each predicate from the IDB a set of statements, which constitute

1. the set of just those statements, provable

from the EDB by applying the rules from the IDB.
2. pro EDB + IDB minimální model.

Proof: by induction on the order of rules.

Recursive DATALOG

Ex.:

In EDB there is a relation
WORKS_FOR(Name_of_w,Chairman)

SUB_SUP(x,y):-WORKS_FOR(x,y)

SUB_SUP(x,y):-WORKS_FOR(x,z), SUB_SUP(z,y)

SUB_SUP* is a transitive closure of the relation WORKS_FOR*

The following holds:

WORKS_FOR $\subseteq$ SUB_SUP

(WORKS_FOR * SUB_SUP)[1,3] $\subseteq$ SUB_SUP

Recursive DATALOG

$\Rightarrow$ SUB_SUP* is a solution of equation

(WORKS_FOR * SUB_SUP)[1,3] $\cup$
WORKS_FOR = SUB_SUP

More generally:

For IDB there is a system of equations

$E_i(P_1,\ldots,P_n) = P_i \qquad i=1,\ldots,n$

The solution of the system depends on EDB and is its *fixpoint*.

Remark: Since all used operations of A_R are additive, the fixpoint exists and even the least one.

Recursive DATALOG

Algorithm: **(Naive) evaluation**

Input: EDB = $\{R_1,\ldots,R_k\}$, IDB = {rules for $P_1,\ldots,P_n$},

Output: least fixpoint $P_1^*,\ldots,P_n^*$

Method: We use a function *eval*(E) evaluating a relational expression E.

for i:=1 to n do $P_i := \emptyset$;

repeat for i:=1 to n do

$Q_i := P_i$; *{store old values}*

for i:=1 to n do

$P_i := eval(E_i(P_1,\ldots,P_n))$

until $P_i = Q_i$ for all i $\in \langle 1,n \rangle$

Remark: It is so-called Gauss-Seidel method.

Recursive DATALOG

Statement: Evaluating algorithm stops and returns the least fixpoint of the system of datalogical equations.

Proof:

(1) follows from the fact that *eval* is monotonic and P_i* are generated from a finite number of elements.

(2) follows from that P_i* is solution of the system of equations and, moreover, it is a part of each solution for each *i*. It can be proved by induction on the number of iterations. The start is from ∅, which is a part of each solution.

Disadvantages:

- creating duplicate tuples,
- creating unnecessarily large relations, when we want, e.g., only a selection of the tuples from P_i* in the result.

Recursive DATALOG

Method of differences

Idea: in the (k+1). step of the iteration we do not calculate P_i^{k+1}, but $D_i^{k+1} = P_i^{k+1} - P_i^k$, i.e.

$$P_i^{k+1} = P_i^k \cup D_i^{k+1} \text{ and thus}$$

$$P_i^{k+1} = E_i(P_i^{k-1}) \cup E_i(D_i^k),$$

since E_i is additive.

The change of *eval* for P_i is given by on rule:

pincreval$(E_i(\Delta P_1,\ldots,\Delta P_n))$

$$= \cup_{j=1..n}\, eval(E_i(\ldots,P_{j-1},\Delta P_j,P_{j+1},\ldots))$$

Recursive DATALOG

The change of eval for P_i given by s rules:

$increval(P_k;\Delta P_1,\ldots,\Delta P_n))$

$= \cup_{j=1..s}\, pincreval(E_j(\Delta P_1,\ldots,\Delta P_n))$

Ex.:

$increval(S') = \varnothing$

$increval(C) =$

$(F(X1,X)*F(X2,Y)*\Delta S'(X1,X2))[X,Y] \cup$

$(F(X1,X)*F(X2,Y)*\Delta C(X1,X2))[X,Y]$

$increval(R) =$

$\Delta S'(X,Y) \cup (\Delta R(X,Y)*F(Z,Y))[X,Y] \cup$

$(\Delta R(Z,Y)*F(Z,X))[X,Y]$

Recursive DATALOG

Algorithm: **(Seminaive) evaluation**

Input: EDB = $\{R_1,\ldots,R_k\}$, IDB = {rules for $P_1,\ldots,P_n$},

Output: least fixpoint $P_1^*,\ldots,P_n^*$

Method: 1$\times$ use the function *eval* and on differences *increval*

```
for i:=1 to n do
          ΔPi := eval (Ei (∅,…,∅));
repeat  for i:=1 to n do ΔQi  := ΔPi;                    {store old diferences}
        for i:=1 to n do   begin
                  ΔPi  := increval(Ei;(ΔQ1,…,ΔQn, P1,…, Pn))
                  ΔPi  := ΔPi - Pi                       {delete duplicates}
                           end ;
        for i:=1 to n do Pi :=  Pi ∪ ΔPi
        until Δ Pi = ∅ for all i  ∈<1,n>
```

Recursive DATALOG

Statement: The evaluating algorithm stops and

- returns the LFP of the system of datalogical equations,
- LFP corresponds just to those facts, which are provable from EDB by rules from IDB.

Ex.: R(x,y) :- P(x,y)

R(x,y) :- R(x,z), R(z,y)

LFP R* is a solution of equation

$$R(X,Y) = P(X,Y) \cup (R(X,Z)*R(Z,Y))[X,Y] \qquad (*)$$

- if P* = {(1,2), (2,3)}, then

  R* = {(1,2), (2,3), (1,3)} is the LFP, whose elements correspond to all derivable facts,

R* is also a minimal model.

Recursive DATALOG

- If (1,1) $\in$ R*, then R(1,1) :- R(1,1),R(1,1), so also R* = {(1,1),(1,2), (2,3), (1,3)} is a model and it is a solution of equation (*).
- If (3,1) $\in$ R*, then {(1,2), (2,3), (1,3), (3,1)}

  is not a model and not a solution of the equation (*).
- Let P* = $\emptyset$; R* = {(1,2)}.

  then R* is a model, but it is not a solution the equation (*).

Use of recursive Datalog in web services

Assumption: web sources with querying, which enables to formulate always a subset of conjunctive queries.

Ex.: Amazon – we enter an author name and obtain the list of his/her books. We can not ask for a list of all available books.

Ex.: Travel service with source relations **R**:

flights(start, end), trains(start, end),

buses(start, end), shuttle(start, end)

Use of recursive Datalog in web services

Datalogical program extends possibilities of conjunctive queries by generating views with recursion, e.g. LP

ans(a, b) :- flights(a,c), ind(c,b)

ind(c,b) :- flights(c,b), buses(b, Praha)

ind(c,b) :- flights(c,c'), ind(c',b)

Remark: However, we can not find out from LP anyway whether Prague is accessible from somewhere with air followed by a shuttle service.

Extension of Datalog by negation

Ex.: NSR(x,y) … x and y are relatives, but x is not a sibling of y

NSR(x,y) :- R(x,y), ¬S'(x,y)

NSR* = R* - S'*

or

NSR(X,Y) = R(X,Y) * <u>S'</u>(X,Y), where <u>S'</u> is the complement to a suitable universe.

Approach:

- We allow a negation in bodies of rules, i.e. negative literals between $L_1,\ldots,L_n$
- safe rules must have limited variables, i.e. we forbid variables, which are in a negative literal and are not limited by the original definition.

Extension of Datalog by negation

Problem:

The solution of a logical program does not have to be LFP, but a number of MFPs.

Ex.: BORING(x) :- ¬ INTERESTING(x), MAN(x)

INTERESTING(x) :- ¬ BORING(x), MAN(x)

B(X) = M(X) - I(X)

I(X) = M(X) - B(X)

Solution: Let M = {John},

M1: {BORING* = {John}, INTERESTING* = ∅}

M2: {INTERESTING* = {John}, BORING* = ∅}

Stratified DATALOG$^{\neg}$

- It is not true, that one model is less than the second one,
- There is no model less than M1 or M2

$\Rightarrow$ we have two minimal models

Intuition: a constraint of the negation – if it is applied , then to a known relation, i.e. such relations have to be first defined (maybe recursively) without negation. Then, a new relation can be defined by them without or with negations.

Df.: Definition of a virtual relation S is a set of all rules, which have S in head.

Df.: S occurs in a rule positively (negatively), if it is contained in a positive (negative) literal.

Stratified DATALOG$^{\neg}$

Df: Program P is stratifiable, if there is a partition $P = P_1 \cup \ldots \cup P_n$ (P_i are mutually disjunctive) such that for each i $\in$<1,n> the following holds:

1. If the relational symbol S occurs positively in a rule from P_i, then the definition of S is contained in $\cup_{j \leq i} P_j$
2. If the relational symbol S occurs negatively in a rule from P_i, then the definition of S is contained in $\cup_{j < i} P_j$

   (P_1 can be $\emptyset$)

Df.: Partition $P_1, \ldots, P_n$ is called a stratification P, each P_i is a stratum.

Remark: stratum … layer

strata … layers

Stratified DATALOG$^\neg$

Ex.: Program P(x) :- ¬ Q(x) (1)

R(1) (2)

Q(x) :- Q(x), ¬ R(x) (3)

is stratifiable. Stratification: {(2)} ∪ {(3)} ∪ {(1)}

Program P(x) :- ¬ Q(x)

Q(x) :- ¬ P(x)

is not stratifiable.

Df.: Let (U,V) is an edge in a dependency graph. (U,V) is positive (negative), if there is a rule V:- … U … and U occurs there positively (negatively).

Remark: An edge can be positive and negative as well.

Stratified DATALOG$^{\neg}$

Statement: Program P is stratifiable if and only if its dependency graph contains no cycle with a negative edge.

Proof: $\Rightarrow$ each virtual relation P has assigned the index of stratum, in which it is defined. Thus, (P,Q) is positive $\Rightarrow$ index(P) $\leq$ index(Q)

(P,Q) is negative $\Rightarrow$ index(P) < index(Q)

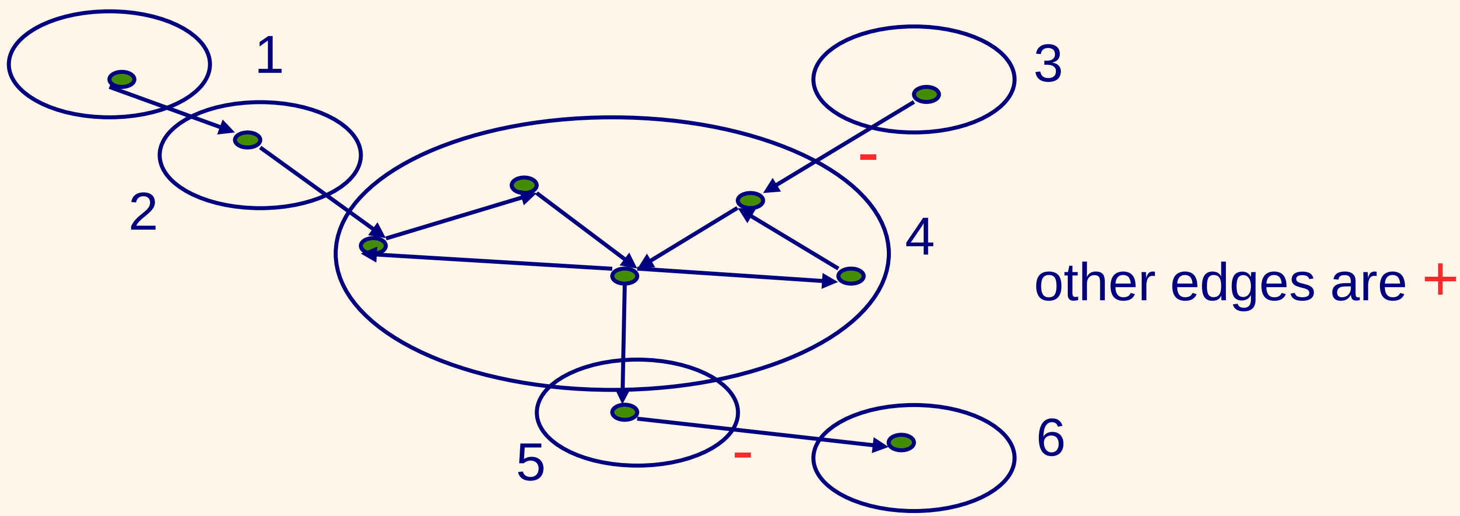
If there was a cycle with a negative edge, there would be a node X, where index(X) < index(X), which is contradiction.

$\Leftarrow$ We find strongly connected components in the dependency graph, then perform the graph's condensation, which is acyclic, and assign a topological ordering of components.

Stratified DATALOG$^{\neg}$

Each component defines one stratum, ordering of components defines their numbering. Since negative edges are at most between components, the rules associated to a component create a stratum.

Ex.:



Stratified DATALOG$^{\neg}$

Assumptions: rules are safe, rectified.

adom … union of constants from EDB and IDB

$\neg$ Q(x_1,…,x_n) is transformed to (*adom* ×...× *adom*) - Q*

Algorithm: **Evaluation of a stratifiable program**

Input: EDB = {R_1,…,R_k}, IDB = {rules for P_1,…,P_n},

Output: minimal fixpoint P_1*,…,P_n*

method: Find a stratification of the program; calculate *adom*;

for i:=1 to s do   *{s strata}*

begin   {for stratum *i* there are relations calculated from strata *j*, where *j*<*i*}

  if Q in stratum *i* is positive then use Q;

  if Q in stratum *i* is is negative then use *adom*^{n} - Q;

  use algorithm for calculation of LFP

end

Stratified DATALOG¬

Statement: Evaluating algorithm stops and returns a MFP of the system of datalogical equations.

Proof: FP follows by induction on the number of strata.

Remark: LP of the stratified DATALOG¬ can have more MFPs.

Stratified DATALOG$^{\neg}$

EDB: Parts(part, subpart, quantity)

tricycle	bike,	3
tricycle	frame	1
frame	saddle	1
frame	pedal	2
bike	rim	1
bike	tire	1
tire	valve	1
tire	inner tube	1

IDB

Large(P) :- Parts(P,S,Q), Q > 2

Small(P) :- Parts(P,S,Q), $\neg$ Large(P)

Stratification and resulting MFP:

Stratum 0: Parts

Stratum 1: Large   Large = {tricycle}

Stratum 2: Small   Small = {frame, bike, tire}

But: relations Small={tricycle, frame, bike, tire}, Large={} provide other MFP of this program, although it is not the result of a stratified evaluation.

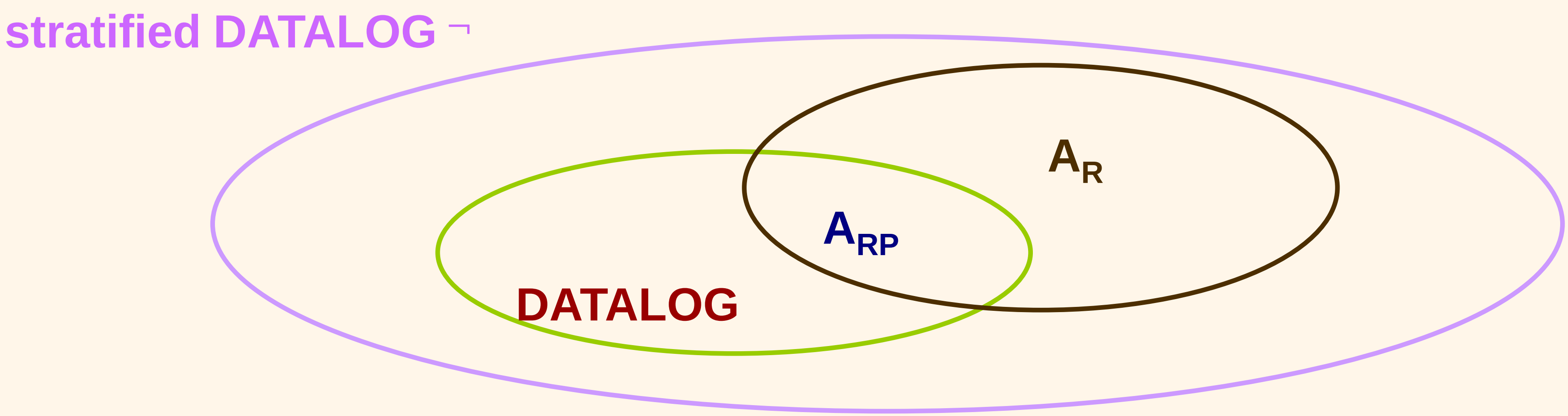
Stratified DATALOG$^{\neg}$

Remark: Stratifiable program has generally more stratifications. They are equivalent, i.e., their evaluation leads to the same MFP (Apt, 1986).

Statement: Non-recursive Datalog programs express just those queries, which are expressible by a monotonic subset of A_R.

Remark: positive relational algebra A_{RP} $\{\times, \cup, [\ ], \varphi\}$.

Stratified DATALOG¬



Relational algebra and DATALOG$^{\neg}$

Statement: Non-recursive DATALOG$^{\neg}$ programs express just those queries, which are expressible in A_R.

Proof: $\Leftarrow$ by induction on the number of operators in E

1. $\emptyset$ of operators: $E \equiv R$ R is from EDB

   $E \equiv$ constant relation

   then for each tuple add $p(a_1,\ldots,a_n)$ into EDB, nothing into IDB.

2. $E \equiv E_1 \cup E_2$

   By induction hypothesis, there are programs for E_1 and E_2 (associated predicates are e_1 and e_2)

   $e(x_1,\ldots,x_n)$:- $e_1(x_1,\ldots,x_n)$

   $e(x_1,\ldots,x_n)$:- $e_2(x_1,\ldots,x_n)$

Relational algebra and DATALOG$^{\neg}$

3. $E \equiv E_1 - E_2$

   $e(x_1,\ldots,x_n)$:- $e_1(x_1,\ldots,x_n)$, $\neg$ $e_2(x_1,\ldots,x_n)$

4. $E \equiv E_1[i_1,\ldots,i_k]$

   $e(x_{i1},\ldots,x_{ik})$:- $e_1(x_1,\ldots,x_n)$,

5. $E \equiv E_1 \times E_2$

   $e(x_1,\ldots,x_{n+m})$:- $e_1(x_1,\ldots,x_n)$, $e_2(x_{n+1},\ldots,x_{n+m})$

5. $E \equiv E_1(\varphi)$

   $e(x_1,\ldots,x_n)$:- $e_1(x_1,\ldots,x_n)$, $x_{ij} = x_{ik}$ or $x_{ij} = a$

$\Rightarrow$ from non-recursiveness: topological ordering + $adom^n - Q^*$ for negation. For each *P* defined in IDB it is possible to construct an expression in A_R . By substitutions (according to ordering) we obtain relational expressions depending only on relations from EDB.

Relational algebra and DATALOG$^{\neg}$

Ex.: *Construction of LP from a relational expression*

CAN_BUY(X,Y) ≡

IS_LIKED(X,Y) - (DEBTOR(X) × IS_LIKED(X,Y)[Y])

EDB: IS_LIKED(X,Y) … person X likes the thing Y

DEBTOR(X) … person X is a deptor

denote DEBTOR(X) × IS_LIKED(X,Y)[Y] as D_A_COUPLE(X,Y).

Then a datalogical program for CAN_BUY is:

IS_ADMIRED(y) :- IS_LIKED(x,y)

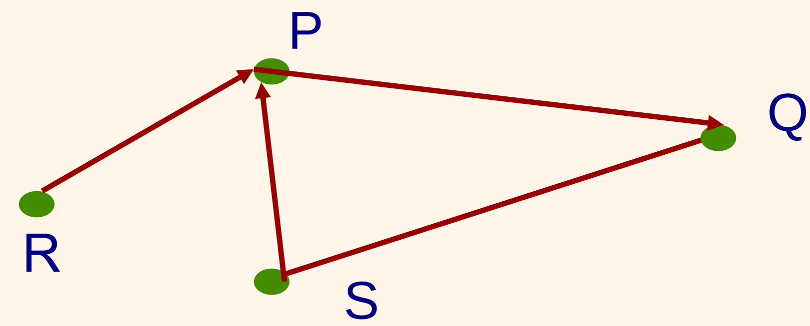
D_A_COUPLE(x,y):-DEBTOR(x), IS_ADMIRED(y)

CAN_BUY(x,y) :- IS_LIKED(x,y), ¬ D_A_COUPLE(x,y)

Relational algebra and DATALOG$^{\neg}$

Ex.: *Construction of a relational expression from LP*

EDB: R^*, S^*, *adom* ≡ R[X] ∪ R[Y] ∪ S



P(x) :- R(x,y), ¬S(y)

Q(z) :- S(z), ¬P(z)

P(X) ≡ (R(X,Y) * {*adom* - S}(Y))[X]

Q(Z) ≡ S(Z) * {*adom* - P}(from) ≡ (S ∩ {*adom* - P})(Z)

Since S ⊂ *adom*, salary Q(Z) ≡ S(Z) - P(Z). After substitution of P

Q(Z) ≡ S(Z) - (R(Z,Y) * {*adom* - S}(Y))[Z]

Remark: *adom* can be replaced by R[Y]

Closed World Assumption (1)

Remark: logical program leads to one resulted relation.

More generally: more (independent) relations $\Rightarrow$ more relational expressions

Ex.: $S'(y,w) := F(x,y), F(x,w), y \neq w$

If F^* is such, that it can not be inferred S´(Moore, Bond), then can be declared $\neg$S´(Moore, Bond)

Remark: It is not proof!

Df.: Consider Horn clauses (without $\neg$). Closed World Assumption (CWA) says: whenever the fact $R(a_1,\ldots,a_k)$ is not derivable from EDB and rules, then $\neg R(a_1,\ldots,a_k)$.

Remark: CWA is a metarule for deriving negative information.

Notation: $\models_{CWA}$

Closed World Assumption (2)

Assumptions for use of CWA:

(1) different constants do not denote the same object

Ex.: F(Flemming, Bond), F(Flemming, 007) $\Rightarrow$ S'(Bond, 007)

If Bond and 007 are names of the same agent, we obtain nonsense

(2) Domain is closed (constants from EDB+IDB)

Ex.: Otherwise, it could be not deduced $\neg$S´(Bond,007);

(they could have his father "out of" database).

Statement: (about CWA consistency): Let *E* is a set of facts from EDB, *I* is a set of facts derivable by the datalogical program IDB$\cup$EDB, *J* is a set of facts the form $\neg$ R($a_1,\ldots,a_k$) , where R is a predicate symbol from IDB$\cup$EDB and R($a_1,\ldots,a_k$) is not in *I* $\cup$ *E*. Then $I \cup E \cup J$ is logically consistent.

Closed World Assumption (3)

Proof: Let $K = I \cup E \cup J$ is not consistent. $\Rightarrow \exists$ rule p(...):- q_1(...),...,q_k(...) and a substitution such that facts on the right side of the rule are in K and derived fact is not in K. Since facts from right side are positive literals, they are from $I \cup E$ and not from J. But then the literal from the rule head has to be from I (is derivable by LFP), that is a contradiction.

Remark: DATALOG$^\neg$ can not be built on CWA.

Ex.: Consider the program

LP: BORING(Emil) :- ¬INTERESTING(Emil)

i.e. ¬INTERESTING(Emil) $\Rightarrow$ BORING(Emil) that is $\Leftrightarrow$

INTERESTING(Emil) $\vee$ BORING(Emil) and therefore neither INTERESTING(Emil) nor BORING(Emil) can be derivable from LP.

Closed World Assumption (4)

LP $\models^{CWA}$ $\neg$ INTERESTING(Emil)

LP $\models^{CWA}$ $\neg$ BORING(Emil)

But no model of LP can contain

$\{\neg$ INTERESTING(Emil),$\neg$BORING(Emil)$\}$

$\Rightarrow$ DATALOG$^{\neg}$ is not consistent with CWA.

Remark: LP has two minimal models:

$\{$BORING(Emil)$\}$ and $\{$INTERESTING (Emil)$\}$

Stratification solves the example naturally:

$EDB_{LP} = \emptyset$

first, INTERESTING is calculated, that is $\emptyset$, then BORING= $\{$Emil$\}$,

i.e., the minimal model $\{$BORING(Emil)$\}$ is chosen.

Closed World Assumption (5)

Consider the program

P': INTERESTING(Emil) :- $\neg$ BORING(Emil)

i.e. $\neg$ BORING(Emil) $\Rightarrow$ INTERESTING(Emil) that is
$\Leftrightarrow$ INTERESTING(Emil) $\vee$ BORING(Emil)

Stratification will chose the model
$\{$INTERESTING(Emil)$\}$

Deductive databases (1)

Informally: EDB $\cup$ IDB $\cup$ IC

Discusion of clauses: clause is universally quantified disjunction of literals

$\neg L_1 \vee \neg L_2 \vee \ldots \vee \neg L_k \vee K_1 \vee K_2 \vee \ldots \vee K_p \qquad (\Leftrightarrow)$

$L_1 \wedge L_2 \wedge \ldots \wedge L_k \Rightarrow K_1 \vee K_2 \vee \ldots \vee K_p$

Remark: in Datalog p=1

(i) k=0, p=1:

facts, e.g., emp(George), earns(Tom,8000)

unrestricted clauses, e.g. likes(Good,x)

(ii) k=1, p=0:

negative facts, e.g. $\neg$ earns(Eduard,8000)

IC, e.g., $\neg$ likes(John,x)

Deductive databases (2)

(iii) $k>1$, $p=0$:

IC, e.g. $\forall x\ (\neg\ man(x) \vee \neg\ woman(x))$

(iv) $k>1$, $p=1$: this is a Horn clause, i.e.,

IC or a deductive rule

(v) $k=0$, $p>1$:

disjunctive information, e.g. $man(x) \vee woman(x)$,

$earns(Eda,8000) \vee earns(Eda,9000)$

(vi) $k>0$, $p>1$:

IC or definition of undeterminate data, e.g.,

$parent(x,y) \Rightarrow father(x,y) \vee mother(x,y)$

(vii) $k=0$, $p=0$:

empty clause (should not be a part of DB)

Deductive databases (3)

df.: Definite *deductive database* is a set clauses, which are neither of type (v) nor (vi). Database containing (v) or (vi) is indefinite.

Definite deductive DB can be understood as a couple

1. theory T, which contains special axioms:
   - facts (associated to tuples from EDB)
   - axioms about elements and facts:
     - completeness (no other facts hold than those from EDB and those derivable by rules)
     - domain closure axiom
     - unique names axiom
   - set of Horn clauses (deductive rules)

Deductive databases (4)

CWA can be used for definite deductive DB.

Remark: this eliminates to need to use axioms of completeness and axiom of unique names $\Rightarrow$ more simple implementation

Statement: Definite deductive DB is consistent.

- answer to a query $Q(x_1,...,x_k)$ in a deductive DB is a set of tuples $(a_1,...,a_k)$ such, that

  $T \models Q(a_1,...,a_k)$,

- deductive database fulfils IC iff $\forall\, c \in IC\ T \models c$.

Remark: if a formal system is correct and complete, then $\vdash$ is the same as $\models$.

Correctness of IS (1)

DB vs. real world (object world)

Requirements:

- consistency

  it is not possible to prove that w and ¬ w

- correctness in the object world

  database is in accordance to the object world

- completeness

  In the system it is possible to prove, that either w or ¬ w.

Correctness of IS (2)

Ex.: problems related to the object world

Sch1: emp(.), salary(.), earns(.,.)

IC: $\forall$ x (emp(x) $\Rightarrow \exists$ y (salary(y) $\wedge$ earns(x,y))

M1: emp: {George, Charles}, salary: {19500, 16700}

earns: { (George, 19500), (Charles, 16700)},

M2: earns INSERT: (19500, 16700) to earns

Sch2: emp(.), salary(.), earns(.,.)

IC: $\forall$x $\exists$y (emp(x) $\Rightarrow$ earns(x,y))

$\forall$x $\forall$y(earns(x,y) $\Rightarrow$ (emp(x) $\wedge$ salary(y)))

M2 is not a model

Achieving consistency: a model construction

IC (1)

IC as closed formulas.

Problems: consistency
nonredundancy

Ex.: functional dependences

- in the language of 1. order logic

  $\forall a,b,c_1,c_2,d_1,d_2$
  $((R(a,b,c_1,d_1) \wedge R(a,b,c_2,d_2) \Rightarrow c_1 = c_2))$

- in the theory of functional dependencies

  $AB \rightarrow C$

Non-redundancy is investigated by the solution of membership problem.

IC (2)

- general dependences

$\forall y_1,\ldots,y_k \exists x_1,\ldots,x_m((A_1 \wedge\ldots\wedge A_p) \Rightarrow (B_1 \wedge\ldots\wedge B_q))$

where

$k, p, q \geq 1$, $m \geq 0$,

A_i … positive literals with variables from $\{y_1,\ldots,y_k\}$

B_i … equalities or positive literals with variables from $\{y_1,\ldots,y_k\} \cup \{x_1,\ldots,x_m\}$

$m = 0$ … full dependences

$m > 0$ … embedded dependences

IC (3)

Classification of dependencies:

- typed (1 variable is not in more columns)
- full, embedded
- tuple-generating, equality-generating
- functional

  inclusion (generally embedded, untyped)

  template (q=1, B je positive literal)

  ...

General dependences - examples

EMBEDDED, TUPLE-GENERATING

$\forall x\ (emp(x) \Rightarrow \exists y\ (salary(y) \wedge earns\ (x,y))$

FULL, EQUALITY-GENERATING, FUNCTIONAL

$\forall x,y_1,y_2(earns(x,y_1) \wedge earns(x,y_2) \Rightarrow y_1=y_2)$

FULL, TUPLE-GENERATING, INCLUSION

$\forall x,\ z\ (manages(x,z) \Rightarrow emp(x))$

FULL (MORE GENARAL)

$\forall x,y,z\ (earns(x,y) \wedge manages(x,z) \Rightarrow y > 5000)$

EMBEDDED, TUPLE-GENERATING, INCLUSION

$\forall x,\ z\ (manages(x,z) \Rightarrow \exists y\ (solves(x,y)\ ))$

Statements about dependencies (1)

Statement: The best procedure solving the membership problem for typed full dependencies has exponential time complexity.

Remark: Membership problem for full dependences is the same for finite and infinite relations.

Ex.: $\Sigma = \{A \rightarrow B,\ A \subseteq B\}$

τ: $B \subseteq A$

It holds: $\Sigma \models_f \tau$ $\qquad \Sigma \not\models \tau$

e.g., on relation $\{(i+1,i): i \geq 0\}$

Statements about dependencies (2)

Statement: Membership problems for general dependences are not equivalent for finite and infinite relation. Both problems are not solvable.

Statement: Membership problem for FD and ID is not solvable.

Statement: Let Σ contain only FD and unary ID. Then the membership problem for finite and also for infinite relations is solvable in polynomial time.

Statements about dependencies (3)

Conclusion: If the exponential time is still tolerable for today’s and future computers, then full dependences are the broadest class of dependencies usable for deductive databases.

$\Rightarrow$ significant role of Horn clauses in computer science.

Pessimistic view:

- Generally, completeness can not be achieved.
- Generally, consistency can not be achieved.
- Algorithmic complexity can be a real issue. It sometimes can not be improved and often not solved – an associated proof procedure does not exist.

Statements about dependencies (4)

- constraints may make consistence, but associated models do not match real world facts.

Optimistic view:

- Pessimistic results are general. What are the sets of real dependencies?

Query languages - problems

- 1982: Chandra and Harel stated a problem:

  Is there a query language (logic), enabling to express exactly all queries computable in polynomial time (PTIME)?

  Answer: unknown till now.
- 1982: Immerman and Vardi proved, that the extension of the 1. order logic by the operator LFP enables it on the class of all ordered finite structures.
- Another approximation: FP+C (counting operator). It enables catch up PTIME, e.g., on all trees, planar graphs and others.
  - Remark: counting enables to find the number of items satisfying a formula.