Query languages 2 (NDBI006)
Expressive power - part 2

J. Pokorný
MFF UK

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.

<u>for</u> i:=1 <u>to</u> n <u>do</u> $P_i := \emptyset$;

<u>repeat</u> <u>for</u> i:=1 <u>to</u> n <u>do</u>

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

<u>for</u> i:=1 <u>to</u> n <u>do</u>

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

<u>until</u> $P_i = Q_i$ 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 $\emptyset$, 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.

$$P_i^{k+1} = P_i^k \cup D_i^{k+1}$$ 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:

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

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

Ex.:

$\mathit{increval}(S') = \emptyset$

$\mathit{increval}(C) =$

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

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

$\mathit{increval}(R) =$

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

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

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× use the function *eval* and on differences *increval*

```
for i:=1 to n do
          ΔP_i := eval (E_i (∅,…,∅));
repeat  for i:=1 to n do ΔQ_i := ΔP_i;                  {store old diferences}
        for i:=1 to n do  begin
                  ΔP_i := increval(E_i;(ΔQ_1,…,ΔQ_n, P_1,…, P_n))
                  ΔP_i := ΔP_i - P_i                     {delete duplicates}
                          end ;
        for i:=1 to n do P_i := P_i ∪ ΔP_i
        until Δ P_i = ∅ 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), $\neg$S'(x,y)

NSR* = R* - S'*

or

NSR(X,Y) = R(X,Y) * $\underline{S'}$(X,Y), where $\underline{S'}$ 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. 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 $i \in \langle 1,n \rangle$ 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 $P(x) \text{ :- } \neg\, Q(x)$ (1)

$R(1)$ (2)

$Q(x) \text{ :- } Q(x),\ \neg\, R(x)$ (3)

is stratifiable. Stratification: $\{(2)\} \cup \{(3)\} \cup \{(1)\}$

Program $P(x) \text{ :- } \neg\, Q(x)$

$Q(x) \text{ :- } \neg\, P(x)$

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 contain 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)

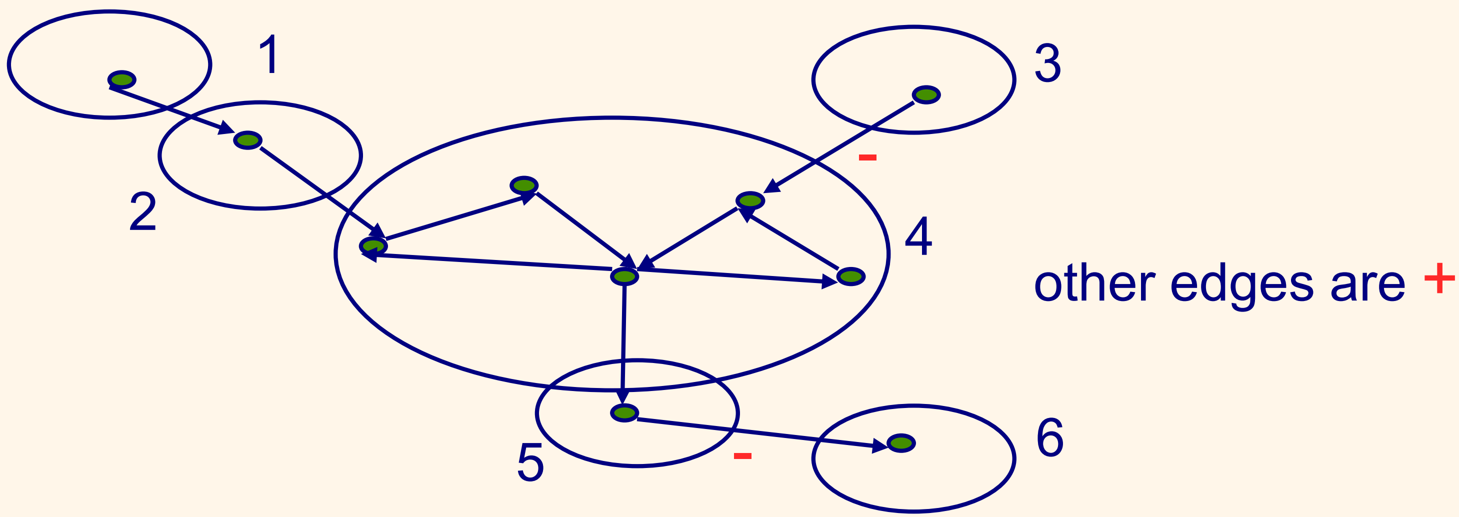
If there was a cycle with a negative edge, there would be a node X, where index(X) < index(X), which is contradiction.

$\Leftarrow$ 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 component 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$^\neg$

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$^\neg$ 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.

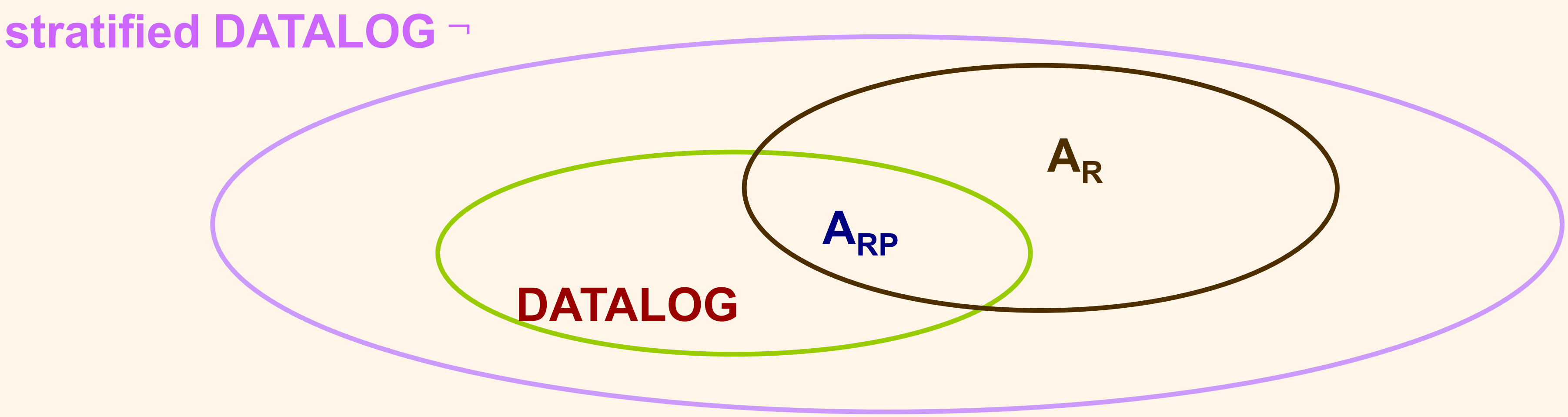
Stratified DATALOG$^{\neg}$

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$^{\neg}$



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

   $E \equiv$ const. 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. $E \equiv E_1 - E_2$

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

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

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

5. $E \equiv E_1 \times E_2$

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

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

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

IS_LIKED(X,Y) - (DEBTOR(X) $\times$ IS_LIKED(X,Y)[Y])

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

DEBTOR(X) … person X is a DEBTOR

denote DEBTOR(X) $\times$ 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)

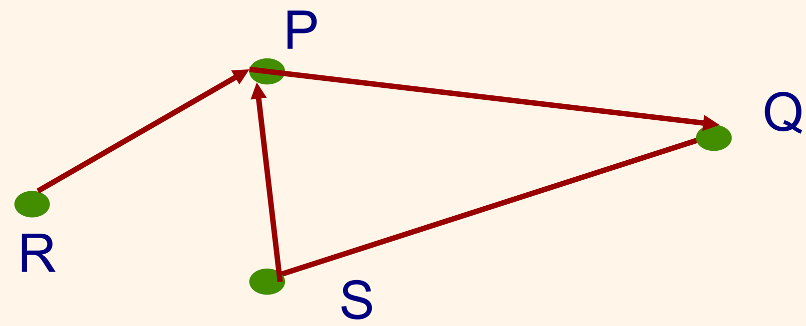
D_A_COUPLE(x,y):-DEBTOR(x), IS_ADMIRED(y)

CAN_BUY(x,y) :- IS_LIKED(x,y), $\neg$ D_A_COUPLE(x,y)

Relational algebra and DATALOG$^{\neg}$

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

EDB: R^*, S^*, *adom* $\equiv R[X] \cup R[Y] \cup S$



$P(x)$:- $R(x,y)$, $\neg S(y)$

$Q(z)$:- $S(z)$, $\neg P(z)$

$P(X) \equiv (R(X,Y) * \{adom - S\}(Y))[X]$

$Q(Z) \equiv S(Z) * \{adom - P\}(\mathrm{from}) \equiv (S \cap \{adom - P\})(\mathrm{from})$

Since $S \subset$ *adom*, salary $Q(Z) \equiv S(Z) - P(Z)$. After substitution of P

$Q(Z) \equiv S(Z) - (R(Z,Y) * \{adom - S\}(Y))[\mathrm{from}]$

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: $\models_{\text{CWA}}$

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 not deduce $\neg S'(Bond,007)$;

(they could have his father "except" 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 facts are 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) :- $\neg$INTERESTING(Emil)

i.e. $\neg$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, it calculates INTERESTING, that is $\emptyset$, then BORING= $\{$Emil$\}$,

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

Closed World Assumption (5)

Consider 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: p=1 in Datalog

(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

(in) 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 uncertain data, e.g. father

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

(vii) k=0, p=0:

empty clauses (should not be a part of DB)

Deductive databases (3)

df.: Definite *deductive database* is a set clauses, which are neither of type (in) 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 $Q(x_1,\ldots,x_k)$ in a deductive DB is a set of tuples $(a_1,\ldots,a_k)$ such, that

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

- 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 $\neg$ 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 $\neg$ 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: 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 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: $\Sigma \models_f \tau$ $\quad\quad\quad$ $\Sigma \not\models \tau$

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 contains 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.