Deductive Databases: Topics

- The logic of query languages
- Bottom-up Semantics
- Top-down Semantics
- Summary

1

Introduction

- First-order logic provides a conceptual foundation for relational query languages
- Relational calculus (RC): logic-based model for declarative query languages
- Relational algebra (RA): operational equivalent of RC
- Safe queries in RC can be transformed into equivalent RA expressions and vice versa
- Transformation of RC into RA: first step in efficient query implementation and optimization
- RC has limited expressive power and cannot express many important queries, e.g., transitive closures and generalized aggregates
- Need of more powerful logic-based languages that subsume RC

2

Datalog

- DB is viewed as a set of facts, one for each tuple in corresponding table of relational DB
- Name of relation becomes predicate name of the fact

student

Name	Major	Year
Joe Doe	cs	senior
Jim Jones	cs	junior
Jim Black	ee	junior

took

Name	Course	Grade
Joe Doe	cs123	2.7
Jim Jones	cs101	3.0
Jim Jones	cs143	3.3
Jim Black	cs143	3.3
Jim Black	cs101	2.7

student('Joe Doe',cs,senior)
student('Jim Jones',cs,junior)
student(Jim Black',ee,junior)

took('Joe Doe',cs123,2.7)
took('Jim Jones',cs101,3.0)
took('Jim Jones',cs143,3.3)
took('Jim Black',cs143,3.3)
took('Jim Black',cs101,2.7)

3

Datalog, cont.

- Fact: logical predicate having only constants (no variables)
- Conventions
  - constants: tokens begining with lowercase characters or numbers, tokens in quotes
  - variables: tokens begining with uppercase characters
- Rules : main construct of Datalog programs

firstReq(Name) ← student(Name,Major,junior),
took(Name,cs101,Grade1),
took(Name,cs143,Grade2).

- Head, body, goals of rules, arity of predicates
- Commas = logical conjuncts, order of goals is immaterial

4

Datalog, cont.

- Logical disjunct: multiple rules sharing same predicate name and arity

  scndReq(Name) ← student(Name,Major,junior),
  took(Name,cs131,Grade), Grade > 3.0.
  scndReq(Name) ← student(Name,_,junior),
  took(Name,cs151,Grade), Grade > 3.0.

- Major occurs only once, can be replaced with anonymous variable _
- Meaning is independent of order in which rules are listed
- Scope of variables local to rules
- Definition of p : set of rules having as their heads same name p and same arity

Datalog vs. Relational Model

Datalog	Relational Model
Base predicate	Table or relation
Derived predicate	View
Fact	Row or tuple
Argument	Column or attribute

- Extensional database : base predicates
- Intensional database : derived predicates (defined by rules)
- Assumption : base predicates never appear in heads of rules

Datalog Rules

- Rules represent a powerful formalism from both theoretical and practical viewpoints
- Goals in rules can be viewed as search patterns
- A problem can be break up into smaller subproblems, each expressed by simple rules
- Derived relations can be used as goals in rules, as for database relations

  req_cs298(Name) ← firstReq(Name),scndReq(Name).

Query Goals

- Specify which of derived relations must be computed
- Boolean or closed queries: contain no variables
- Their answer is yes or no

  ?firstReq('Jim Black')

- Open queries contain variables

  ?firstReq(X)

- Their answer is a (possibly empty) set of facts satisfying the query

  firstReq('Jim Black')
  firstReq('Jim Black')

- Query goals mix variables and constants in their arguments

Datalog and Negation

- Negation can only be applied to goals of rules

hasTaken(Name,Course) ← took(Name,Course,Grade).
lacks_cs143(Name) ← student(Name,_,junior),
¬hasTaken(Name,cs143).

- A common use of negation is in conjunction with universally quantified queries, often expressed by words such as "each" and "every"
- Example: find senior students who completed all requirements for a cs major

9

Datalog and Universal Quantification

- An universally quantified condition can only be expressed by an equivalent condition with existential quantification and negation
- This transformation requires two steps

(1) Formulate complementary query : find students who did not take some of the courses required for a cs major

reqMissing(Name) ← student(Name,_,senior),req(cs,Course),
¬ hasTaken(Name,Course)

(2) Original query reexpressed as : find senior students who are not missing any requirement for a cs major

allReqSat(Name) ← student(Name,_,senior),
¬ reqMissing(Name).

10

Relational Calculi

- Two types
  - domain relational calculus: variables denote values of attributes
  - tuple relational calculus: variables denote tuples
- Provide a link to commercial database languages: QBE is based on DRC, QUEL and SQL on TRC
- Example: query ?firstReq(N) in DRC and TRC

$$\{(N) \mid \exists G_1(took(N, cs101, G_1)) \wedge \exists G_2(took(N, cs, 143, G_2)) \wedge \exists M(student(N, M, junior))\ \}$$

$$\{(t[1]) \mid \exists u \exists s(took(t) \wedge took(u) \wedge student(s) \wedge t[2] = cs101 \wedge u[2] = cs143 \wedge t[1] = u[1] \wedge s[3] = junior \wedge s[1] = t[1]\ \}$$

- TRC requires explicit statement of equality, in DRC equality denoted by the presence of the same variable in different places

11

Relational Calculi and Datalog

- TRC and DRC are equivalent: mappings transform a formula in one language into an equivalent formula in the other
- Syntactic differences between calculi and Datalog: set definition by abstraction, nesting of parentheses, mixing of conjuntions and disjunctions in the same formula, negation, explicit existential and universal quantifiers
- Example: query ?allReqSat(N) in DRC

$$\{(N) \mid \exists M(student(N, M, junior)) \wedge \forall C(req(cs, C) \rightarrow \exists G(took(N, C, G))\ \}$$

- For each DRC expression there is an equivalent nonrecursive Datalog program. The converse is also true

12

Relational Algebra

- Union : $R \cup S = \{t \mid t \in R \vee t \in S\}$ [a]
- Difference : $R - S = \{t \mid t \in R \wedge t \notin S\}$ [a]
- Cartesian Product:

  $R \times S = \{t \mid (\exists r \in R)(\exists s \in S)(t[1,\ldots,n] = r \wedge t[n+1,\ldots,m] = s)\}$
- Projection: $\pi_L R = \{r[L] \mid r \in R\}$, $L \subseteq \{\$1,\ldots,\$n\}$
- Selection: $\sigma_F(R) = \{t \mid t \in R \wedge F'\}$,
  - F is composed from $\$i\theta C$ and $\$i\theta\$j$ using $\wedge$, $\vee$, and $\neg$
  - F' denotes the formula obtained from F by replacing $\$i$ and $\$j$ by $t[i]$ and $t[j]$

  Example : $\sigma_{\$2=\$3 \wedge \$1=bob} R = \{t \mid t \in R \wedge t[2] = t[3] \wedge t[1] = bob\}$

[a] Relations must be union-compatible

Derived Algebraic Operators

- Join : $R \bowtie_F S = \sigma_{F'}(R \times S)$
  - $F = \$i_1\theta_1\$j_1 \wedge \ldots \wedge \$i_k\theta_1\$j_k$
  - $F' = \$i_1\theta_1\$(m+j_1) \wedge \ldots \wedge \$i_k\theta_1\$(m+j_k)$
- Intersection $R \cap S$ constructed either by
  - equijoin of R and S in every column, and projecting out duplicate columns
  - $R \cap S = R - (R - S) = S - (S - R)$
- Generalized projection: $\pi_L(R)$, L is a list of constants and column names (components may appear more than once)
  - E.g., $\pi_{\$1,c,\$1}(R)$

From Safe Datalog to Relational Algebra

- Datalog, DRC and TRC are declarative logic-based languages, relational algebra is an operator-based language
- Formulas in logical languages can be implemented by transforming them into equivalent RA expressions
- Only safe Datalog can be maped into equivalent RA expressions
- Not a limitation: enforcing safety enables compiler-time detection of rules and queries inadequately specified

Problems with Unsafe Rules

betterGrade(G1) ← took('Joe Doe', cs143,G), G1 > G.

- Infinitely many numbers satisfy condition
- Lack of domain independence, i.e., answer must depend on DB and constants in the query, not on domain of interpretation
  - set of values of G1 depends on domain assumed for numbers
- No RA equivalent
  - only relations are allowed as operands of RA expressions
  - as relations are finite, result of RA expressions is also finite
- Domain independence and finiteness of answers are undecidable even for nonrecursive queries
- Sufficient conditions must be used

Safe Datalog

Inductive definition of safety for a program P

(1) Safe predicates: predicate q of P is safe if

(a) q is a database predicate, or

(b) every rule defining q is safe

(2) Safe variables: variable X in rule r is safe if

(a) X is contained in some positive goal $q(t_1, \ldots, t_n)$, where predicate $q(t_1, \ldots, t_n)$ is safe, or

(b) r contains some equality goal $X = Y$, where Y is safe

(3) Safe rules: rule r is safe if all its variables are safe

(4) Goal $?q(t_1, \ldots, t_n)$ is safe when predicate $q(A_1, \ldots, A_n)$ is safe

17

From Safe Datalog to RA

Mapping a safe nonrecursive Datalog program P into RA

(1) P is transformed into an equivalent program P' not containing any equality goal

$$\mathsf{s(Z,b,W)} \leftarrow \mathsf{q(X,X,Y),p(Y,Z,a),W{=}Z,W{>}24.3.}$$

is translated into

$$\mathsf{s(Z,b,Z)} \leftarrow \mathsf{q(X,X,Y),p(Y,Z,a),W{>}24.3.}$$

(2) Body of a rule r translated into RA expression $Body_r$: Cartesian product of relations in body followed by selection σ_F accounting for equalities and inequalities

$$Body_r = \sigma_{\$1=\$2,\$3=\$4,\$6=a,\$5>24.3}(Q \times P)$$

18

From Safe Datalog to RA, cont.

(3) Each rule r is translated into a generalized projection on $Body_r$, according to the patterns in the head of r

$$S = \pi_{\$5,b,\$5} Body_r$$

(4) Multiple rules with the same head are translated into the union of their equivalent expressions

19

Mapping Rules with Negated Goals

$$r : \ldots \leftarrow \mathsf{b1(a,Y),b2(Y),\neg b3(Y)}$$

- Positive body: negated goal is dropped

$$rp : \ldots \leftarrow \mathsf{b1(a,Y),b2(Y)}$$

- Negative body: remove negation from negated goal

$$rn : \ldots \leftarrow \mathsf{b1(a,Y),b2(Y),b3(Y)}$$

- Both bodies are safe, can be transformed into RA expressions giving $Body_{rp}$ and $Body_{rn}$
- Body expression to be used in step 3 $Body_r = Body_{rp} - Body_{rn}$
- Rules with several negated goals can be translated by repeating this mapping for each negated goal

20

Mapping Rules with Several Negated Goals

- Rules with several negated goals can be translated by repeating this mapping for each negated goal, e.g.,

$$r : \ldots \rightarrow \mathsf{b1(X,Y)}, \neg\mathsf{b2(X)}, \neg\mathsf{b3(Y)}.$$

- Positive body: all negated goals are dropped

$$rp : \ldots \rightarrow \mathsf{b1(X,Y)}.$$

- Negative bodies
  - one for each negated goal
  - add to the positive body one of the negated goals, without negation

$$rn_1 : \ldots \rightarrow \mathsf{b1(X,Y)}, \mathsf{b2(X)}.$$
$$rn_2 : \ldots \rightarrow \mathsf{b1(X,Y)}, \mathsf{b3(Y)}.$$

- Body expression: $Body_{rp} - Body_{rn_1} - \ldots - Body_{rn_k}$

21

Relaxing Safety

- Safety conditions can be relaxed in several ways to improve flexibility and ease-of-use
- One extension: allow existential variables in negated goals, variables are not used anywhere else in the rule
- For example

student(Nme,Yr) ← student(Nme,cs,Yr),¬took(Nme,cs143,G).

can be viewed as a shorthand to

projectTook(Nme,cs143) ← took(Nme,cs143,G).
student(Nme,Yr) ← student(Nme,cs,Yr),¬projectTook(Nme,cs143).

22

Commercial Query Languages

- Goal: simplify DRC and TRC to make them more user-friendly
- QBE is based on DRC, QUEL and SQL on TRC
- Main modification: ensuring that every variable is range quantified, i.e., associated with a relation thus ensuring safety
- Example of transformation of TRC into SQL

$$\{(t[1]) \mid t \in took \exists u \in took \exists s \in student \; (t[2] = cs101 \wedge u[2] = cs143 \wedge t[1] = u[1] \wedge s[3] = junior \wedge s[1] = t[1] \; \}$$

```
SELECT t.Name
FROM took t, took u, student s
WHERE t.Course='cs101' AND u.Course='cs143' AND
        t.Name = u.Name AND s.Year='junior' AND
        s.Name = t.Name
```

23

Universal Quantification in SQL

- EXISTS and ALL are allowed in nested SQL queries
- Universal quantifiers must be expressed using double negation and existential quantifiers

```
SELECT Name
FROM Student
WHERE Year='senior' AND Name NOT IN
        (SELECT S.Name
        FROM student s, req r
        WHERE r.Major='cs' AND s.Year='senior' AND
                NOT EXISTS
                (SELECT t.*
                FROM took t
                WHERE t.Course=r.Course AND t.Name=s.Name
                )
        )
```

24

Beyond SQL

- Excepted set aggregates, the many additional constructs cluttering SQL do not extend its expressive power
- Current practice: procedural languages with embedded SQL
- Impedance mismatch: different data types and computational paradigms
- More powerful query languages would allow a larger portion of the application to be developped in the DB query language
- Result: better data independence and distributed processing
- Datalog provides, in terms of syntax and semantics, a better vehicle for investigating the design of more powerful DB query languages

Recursive Rules

- Bill of materials (BoM): assemblies containing superparts composed of subparts, eventually composed of basic parts

partCost

basicPart	supplier	cost	time
topTube	cinelli	20.00	14
topTube	columbus	15.00	6
downTube	columbus	10.00	6
headTube	cinelli	20.00	14

assembly

part	subpart	qty
bike	frame	1
bike	wheel	2
frame	topTube	1
frame	downTube	1

- To find all subparts of a given part a recursive rule is needed

```
allSubparts(Part,Sub) ← assembly(Part,Sub,_).
allSubparts(Part,Sub) ← allSubparts(Part,Sub1),assembly(Sub1,Sub,_).
```

- First rule: nonrecursive exit rule
- This computes the transitive closure of the aggregation graph
- Transitive closure computations are very common in applications

Example: Basic Subparts

Compute how long it takes to obtain all basic subparts of an assembly

- Find for each part its basic subparts

```
basicSubparts(BasicP,BasicP) ← partCost(BasicP,_,_,_).
basicSubparts(Part,BasicP) ←   assembly(Part,SubP,_),
                               basicSubparts(SubP,BasicP).
```

- For each basic part, find the least time needed for delivery

```
fastest(Part,Time) ← partCost(Part,_,_,Time), ¬faster(Part,Time).
faster(Part,Time) ←  partCost(Part,Sup,_,Time),
                     partCost(Part,Sup1,_,Time1), Time1<Time.
```

- Times required for basic subparts of the given assembly

```
timeForBasic(AssPart,BasicSub,Time) ←
        basicSubparts(AssPart,BasicSub), fastest(BasicSub,Time).
```


Example: Basic Subparts, cont.

- Maximum time required for basic subparts of the given assembly

```
howSoon(AssPart,Time) ←  timeForBasic(AssPart,_,Time),
                         ¬larger(AssPart,Time).
larger(Part,Time) ←  timeForBasic(Part,_,Time),
                     timeForBasic(Part,_,Time1), Time1>Time.
```

- Nonrecursive Datalog with negation can express min and max
- Other aggregates (e.g., count, sum) require stratified Datalog with arithmetic
- Counting the elements in a set modulo an integer does not require arithmetic

Example: Counting Elements

- Determine if a base relation br contains an even number of elements

```
between(X,Z) ← br(X),br(Y),br(Z),X<Y,Y<Z.
next(X,Y) ←     br(X),br(Y),X<Y,¬between(X,Y).
next(nil,X) ←   br(X),¬smaller(X)
smaller(X) ←    br(X),br(Y),Y<X.
even(nil)
even(Y) ←       odd(X),next(X,Y).
odd(Y) ←        even(X),next(X,Y).
brIsEven ←      even(X),¬next(X,Y).
```

- Predicate next sorts elements of br into an ascending chain
- First link of the chain connects nil to the least element in br
- Relies on the assumption that the elements of br are totally ordered by > are can therefore be visited one at a time using this order

Counting with Arithmetics

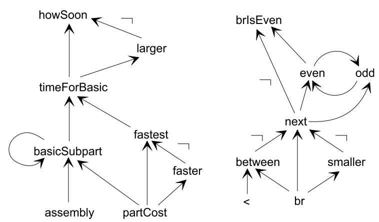
- Counting the number of elements of a base relation br

```
nbElements(0,nil).
nbElements(N,X) ← nbElements(N1,Y), next(Y,X), N=N1+1.
nbElements(N) ← nbElements(N,X), ¬next(X,Y).
```


Stratification

- Predicate dependency graph for a program P, $pdg(P)$
  - Nodes: names of the predicates in P
  - Arc $g \rightarrow h$ if there is a rule r with goal g and head h. If goal is negated, arc is marked as a negative arc
- Nodes and arcs of the strong components of $pdg(P)$ identify recursive predicates and recursive rules of P
- If rule r defines a recursive predicate p, the number of goals in r that are mutually recursive with p is called the rank of r
  - $rank(r) = 0$: exit rule; recursive rule otherwise
  - $rank(r) = 1$: linear rule; nonlinear otherwise
  - $rank(r) = 2$: quadratic, $rank(r) = 3$: cubic

Predicate Dependency Graph



- BoM program: basicSubparts only recursive rule
- Parity query: strong component having as nodes the mutually recursive predicates even and odd
- Both programs are stratifiable: no arc marked with negation belongs to a strong component of the graph (a directed cycle)

Stratifiable Programs

- Non-stratifiable programs are ill-defined from a semantic viewpoint
- Given a statifiable program P, applying topological sorting on $pdg(P)$, the nodes of P can be partitioned into a finite set of strata $1, \ldots, n$ where for each rule $r \in P$, the predicate of the head belongs to a stratum that
  - is $\geq$ to each stratum containing some positive goal of r; and
  - is $>$ than each stratum containing some negated goal of r
- Strata structure the computation: predicates of stratum j are used only after every predicate of lower stratum has been computed
- Strict stratification : every stratum contains either a single predicate or a set of predicates that are mutually exclusive

33

Functors

- Store complex terms and variable-length subrecords in tuples

```
part(202,circle(11),actualKg(0.034)).
part(21,rectangle(10,20),unitKg(2.1)).
partWeigth(No,Kilos) ← part(No,_,actualKg(Kilos)).
partWeigth(No,Kilos) ← part(No,Shape,unitKg(K)),
                       area(Shape,Area),Kilos=K*area.
area(circle(Dmtr),A) ← A=Dmtr*Dmtr*3.14/4.
area(rectangle(Base,Height),A) ← A=Base*Height.
```

- In actual applications functions are used as variable-length subrecords
- Functions can be nested, can be used as discriminants to prescribe different computations

34

Lists

- Functors can be used to generate recursive objects as lists :
  - list(nil) for empty list, list(Head,Tail) for nonempty list
- Most LP languages provide a special notation for lists
- List-based representation of suppliers of topTube

```
partSupList(topTube,[cinelli,columbus,mavic]).
```

- Normalizing a nested relation into a flat relation

```
flatten(P,S,L) ← partSupList(P,[S|L]).
flatten(P,S,L) ← flatten(P,_,[S|L]).
ps(Part,Sup) ← flatten(Part,Sup,_).
```

- Applying these rules yield

```
ps(topTube,cinelli).
ps(topTube,columbus).
ps(topTube,mavic).
```

35

Lists, cont.

- Constructing a nested relation from a normalized relation

```
between(P,X,Z) ← ps(P,X),ps(P,Y),ps(P,Z),X<Y,Y<Z.
smaller(P,X) ← ps(P,X),ps(P,Y),Y<X.
nested(P,[X]) ← ps(P,X),¬smaller(P,X).
nested(P,[Y|[X|W]]) ← nested(P,[X|W]),ps(P,Y),X<Y, ¬between(P,X,Y).
psNested(P,W) ← nested(P,W),¬nested(P,[X|W]).
```

36

Syntax of First-Order Logic

Alphabet consists of

- Constants
- Variables
- Functions $f(t_1, \ldots, t_n)$, f is an n-ary functor and $t_1, \ldots, t_n$ are terms
- Predicates
- Connectives: $\vee$, $\wedge$, $\neg$, $\leftarrow$, $\rightarrow$, $\leftrightarrow$
- Quantifiers: $\forall$, $\exists$
- Parentheses and punctuation symbols, used to avoid ambiguities

Syntax of First-Order Logic, cont.

- A term is defined inductively by
  - a variable is a term
  - a constant is a term
  - $f(t_1, \ldots, t_n)$ is a term if f is an n-ary functor and $t_1, \ldots, t_n$ are terms
- Well-formed formulas (WFF)
  - If p is an n-ary predicate and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an atomic formula, or an atom
  - If F and G are formulas then so are $\neg F$, $F \vee G$, $F \wedge G$, $F \leftarrow G$, $F \rightarrow G$, $F \leftrightarrow G$,
  - If F is a formula and x is a variable then $\forall x(F)$ and $\exists x(F)$ are formulas. When so, x is said to be quantified in F
- Ground terms, atoms, and formulas : contain no variables

Syntax of First-Order Logic, cont.

- Well-formed formulas

$$\begin{array}{l} \exists G_1(took(N, cs101, G_1)) \wedge \exists G_2(took(N, cs143, G_2)) \wedge \\ \quad \exists M(student(N, M, junior)) \\ \exists N, \exists M(student(N, M, junior) \wedge \forall C(req(cs, C) \rightarrow \\ \quad \exists G(took(N, C, G)))) \end{array}$$

- Closed WFF F : every variable in F is quantified
- Free variable in F: variable that is not quantified in F
- Clause: closed WFF of the form

$$\forall x_1, \ldots, \forall x_s(A_1 \vee \ldots \vee A_k \vee \neg B_1 \vee \ldots \vee \neg B_n)$$

  A_i, B_j are atoms, $x_1, \ldots, x_s$ are all variables in these atoms

Syntax of First-Order Logic, cont.

- Definite clause: one positive atom and zero or more negated atoms

$$\forall x_1, \ldots, \forall x_s(A \vee \neg B_1 \vee \ldots \vee \neg B_n)$$

- Since $F \leftarrow G \equiv F \vee \neg G$ this can be rewritten

$$A \leftarrow B_1, \ldots, B_n$$

  where A is the head and $B_1, \ldots, B_n$ is the body of the rule
- Unit clause: definite clause with empty body
  - written A. instead of $A \leftarrow$.
- Fact: unit clause without variables

```
loves(X,X).          loves(marc,mary).
loves(mary,tom).     loves(marc,tom).
```

- Positive logic program: set of definite clauses

Semantics

- Alternative semantics for positive logic programs are equivalent
- Semantics for more general programs (e.g. with negation) is more complex
- Model-theoretic semantics: declarative meaning of a program
- Fixpoint semantics: bottom-up implementation of deductive DBs
- Proof-theoretic semantics: SLD-resolution and top-down execution

Interpretation of a Program P

- Defined with respect to constant symbols, function symbols, and predicate symbols of P
- More generally, for a first-order language L
- Universe (domain) of interpretation: nonempty set of elements U
- Interpretation of L consists of
  - For each constant in L, an assignment of an element in U
  - For each n-ary function in L, the assignment of a mapping from U^n to U
  - For each n-ary predicate q in L, the assignment of a mapping from U^n intro true, false (i.e., a relation on U^n)

Herbrand Interpretations

- Definite clause languages and programs: sufficient to consider interpretations where constants and functions represent themselves
- Functors viewed as variable-length subrecords
- Herbrand universe for L (U_L): set of terms recursively constructed by letting arguments of functions to be constants in L or elements in U_L
- Herbrand base of L: set of atoms that can be built by assigning elements of U_L to the arguments of the predicates
- Herbrand interpretation (HI): assign to each n-ary predicate q, a relation Q of arity n, where $q(a_1, \ldots, a_n)$ is true iff $a_1, \ldots, a_n \in Q$, $a_1, \ldots, a_n \in U_L$
- Alternatively: a Herbrand interpretation of L is a subset of the Herbrand base of L

Herbrand Interpretations, cont.

- For a program P, Herbrand universe U_P and Herbrand base B_P defined as U_L and B_L of the language L that has as constants, functions, and predicates those appearing in P
- In the program

```
anc(X,Y) ← parent(X,Y).
anc(X,Z) ← anc(X,Y),parent(Y,Z).
parent(X,Y) ← father(X,Y).
parent(X,Y) ← mother(X,Y).
mother(anne,silvia). mother(anne,marc).
```

  - $U_P = \{\mathsf{anne}, \mathsf{silvia}, \mathsf{marc}\}$
  - $B_P = \{\mathsf{parent}(x,y) \mid x,y \in U_P\} \cup \{\mathsf{father}(x,y) \mid x,y \in U_P\} \cup \{\mathsf{mother}(x,y) \mid x,y \in U_P\} \cup \{\mathsf{anc}(x,y) \mid x,y \in U_P\}$
  - $|B_P| = 4 \times 3 \times 3 = 36$
  - $2^{|B_P|} = 2^{36}$ Herbrand interpretations

Herbrand Interpretations, cont.

- Program P with an infinite B_P and an infinite number of interpretations

$$\mathsf{p(f(X))} \leftarrow \mathsf{q(X).}$$
$$\mathsf{q(a)} \leftarrow \mathsf{p(X).}$$

  – $U_P = \{a, f(a), \ldots, f^n(a), \ldots\}$, where $f^0(a), f^1(a), f^2(a) \ldots$ stand for $a, f(a), f(f(a)), \ldots$
  – $B_P = \{p(f^n(a)) \mid n \geq 0\} \cup \{q(f^m(a)) \mid m \geq 0\}$

Models of a Program

- $ground(r)$: set of ground instances of r, i.e., rules obtained by assigning values from U_P to variables in r
- For parent(X,X) $\leftarrow$ mother(X,X)., $|U_P| = 3$, $|ground(r)| = 9$

parent(anne,anne) $\leftarrow$ mother(anne,anne).
parent(anne,marc) $\leftarrow$ mother(anne,marc).
...
parent(silvia,silvia) $\leftarrow$ mother(silvia,silvia).

- Ground version of a program P:

$$ground(P) = \{ground(r) \mid r \in P\}$$

- If I is an interpretation, every ground atom $a \in I$ is said to be true (or satisfied), if $a \notin I$ is said to be false (or not satisfied)
- A formula consisting of ground atoms and logical connectives is defined as true or false according to the rules of propositional logic

Models of a Program, cont.

- A rule $r \in P$ is true in interpretation I if every instance of r is satisfied in I
- Model for P: interpretation making true all rules of P
- I is a model for P iff it satisfies all the rules in $ground(P)$
- Interpretations and models for the example
  – $I_1 = \emptyset$ is not a model: facts are not satisfied
  – $I_2 = \{mother(anne, silvia), mother(anne, marc)\}$ is not a model
  – $I_3 = \{mother(a, s), mother(a, m), parent(a, s), parent(a, m), anc(a, s), anc(a, m)\}$ is a model
  – $I_4 = I_3 \cup \{anc(silvia, marc)\}$ is also a model but it is not a minimal one

Properties of Models

- If M_1, M_2 are models for P, then $M_1 \cap M_2$ is also a model for P
- A model M for a program P is minimal if there is no other model M' of P where $M' \subset M$
- A model M for a program P is its least model if $M' \supseteq M$ for every model M' of P
- Every positive program has a least model
- Least model of a program P (M_P): logic-based declarative definition of its meaning
- Need: constructive semantics for realizing minimal model semantics

Fixpoint-Based Semantics

- Views rules as constructive derivation patterns: from the tuples satisfying the goals in a rule the head atoms are constructed
- Relational algebra can be used for such a mapping from the body relations to the head relations
  - parent can be derived through union
  - grandParent from these
  - anc is both the arugment and the result of the RA expression
- Fixpoint equation : $x = T(x)$ where T is a mapping $U \to U$
- Fixpoint for T: a value x that satisfies this equation
- For an arbitrary T there might be zero or more fixpoints

49

Immediate Consequence Operator

- Mapping T_P, the immediate consequence operator for P

  $T_P = \{A \in B_P \mid \exists r : A \leftarrow A_1, \ldots, A_n \in ground(P), \{A_1, \ldots, A_n\} \subseteq I\}$
- Is a mapping from HIs of P to HIs of P
- For the program P

  anc(X,Y) ← parent(X,Y).
  anc(X,Z) ← anc(X,Y),parent(Y,Z).
  parent(X,Y) ← father(X,Y).
  parent(X,Y) ← mother(X,Y).
  mother(anne,silvia). mother(anne,marc).

  - $I = \{anc(anne, marc), parent(marc, silvia)\}$
  - $T_P = \{anc(marc, silvia), anc(anne, silvia), mother(anne, silvia), mother(anne, marc)\}$
- In addition to the atoms derived from the applicable rules, T_P returns also the facts and the ground instances of unit clauses

50

Immediate Consequence Operator

- For the program P

  p(f(X)) ← q(X).
  q(a) ← p(X).

  then $U_P = \{a, f(a), \ldots, f^n(a), \ldots\}$
  - If $I = \{p(a)\}$, then $T_P(I) = \{q(a)\}$
  - If $I_1 = \{p(x) \mid x \in U_P\} \cup \{q(y) \mid y \in U_P\}$, then $T_P(I_1) = \{q(a)\} \cup \{p(f^n(a)) \mid n \geq 1\}$
  - If $I_2 = \emptyset$, then $T_P(I_2) = \emptyset$
  - If $I_3 = T_P(I_1)$ then $T_P(I_3) = \{q(a)\} \cup \{p(f(a))\}$

51

Fixpoint Semantics

- A program P defines the fixpoint equation $I = T_P(I)$ over HIs
- A fixpoint equation may have zero, one, or several solutions
- Equation is over HIs, i.e., subsets of B_P partially ordered with $\subseteq$
- $(2^{|B_P|}, \subseteq)$
  - partial order (transitive, reflexive, and antisymmetric)
  - lattice where $I_1 \cap I_2$ and $I_1 \cup I_2$ define the *lub* and the *glb*
- Complete lattice: given a set of elements in $2^{|B_P|}$ there exists the $\cup$ and $\cap$ of such a set, even it it contains infinitely many elements
- T_P for definite clause programs is monotonic, i.e., if $N \leq M$, then $T_P(N) \leq T_P(M)$
- If P is a definite clause program, then
  - there always exists a least fixpoint for T_P, denoted $lfp(T_P)$
  - $M_P = lfp(T_P)$

52

Powers of T_P

- For positive programs, $lfp(T_P)$ computed by repeated applications of T_P
- $T_P^{\uparrow n}$: n^{th} power of T_P is defined by

$$T_P^{\uparrow 0}(I) = I$$
$$\ldots$$
$$T_P^{\uparrow n+1}(I) = T_P(T_P^{\uparrow n})$$

- With ω denoting the first limit ordinal

$$T_P^{\uparrow \omega}(I) = \bigcup\{T_P^{\uparrow n}(I) \mid n \geq 0\}$$

- For a definite clause program P, $lfp(T_P) = T_P^{\uparrow \omega}(\emptyset)$
- This gives a simple algorithm for computing $lfp(T_P)$
  - Starting from the bottom
  - Iterating the application of T ad infinitum or until no new atoms are obtained and the $(n+1)^{th}$ power equals the n^{th} power

Top-Down Execution

- Each goal in a rule body viewed as a call to a procedure defined by other rules in the same stratum or in lower strata

```
part(202,circle(11),actualKg(0.034)).
part(21,rectangle(10,20),unitKg(2.1)).
partWeigth(No,Kilos) ← part(No,_,actualKg(Kilos)).
partWeigth(No,Kilos) ← part(No,Shape,unitKg(K)),
                        area(Shape,Area),Kilos=K*area.
area(circle(Dmtr),A) ← A=Dmtr*Dmtr*3.14/4.
area(rectangle(Base,Height),A) ← A=Base*Height.
```

- A and Area $\approx$ formal and actual parameters in procedural languages
- Arguments can be complex $\Rightarrow$ passing of parameters through unification

Unification

- Substitution θ: finite set $\{v_1/t_1, \ldots, v_n/t_n\}$, each v_i is a distinct variable, t_i term distinct from v_i
- Ground substitution if every t_i is a ground term
- If E is a term and θ a substitution for variables of E, then $E\theta$ is the result of applying θ to E
- E.g., $E = p(x, y, f(a))$, $\theta = \{x/b, y/x\} \Rightarrow E\theta = p(b, x, f(a))$
- Variables that are not part of the substition left unchanged
- Composition $\theta\delta$ of $\theta = \{u_1/s_1, \ldots, u_m/s_m\}$ and $\delta = \{v_1/t_1, \ldots, v_n/t_n\}$: substitution obtained from $\{u_1/s_1\delta, \ldots, u_m/s_m\delta, v_1/t_1, \ldots, v_n/t_n\}$ by deleting any $u_i/s_i\delta$ for which $u_i = s_i\delta$ and deleting any v_j/t_j for which $v_j \in \{u_1, \ldots, u_m\}$
- E.g., $\theta = \{x/f(y), y/z\}$, $\delta = \{x/a, y/b, z/y\}$, $\theta\delta = \{x/f(b), z/y\}$

Unifiers

- Substitution θ is a unifier for terms A et B if $A\theta = B\theta$
- Most general unifier (mgu): unifier for two terms such that for each other unifier γ there is a substitution δ such that $\gamma = \theta\delta$
- E.g., $p(f(x), a)$ and $p(y, f(w))$ not unifiable
- E.g., $p(f(x), z)$ and $p(y, a)$ unifiable since $\delta = \{y/f(a), x/a, z/a\}$ is a unifier. Mgu is $\theta = \{y/f(x), z/a\}$
- There are efficient algorithms to perform unification: they return either a mgu or reports that none exists
- Given $r : A \leftarrow B_1, \ldots, B_n$ and $\leftarrow g$ (r and g have no variables in common), if there is an mgu δ for A and g, the resolvent of r and g is $\leftarrow B_1\delta, \ldots, B_n\delta$

SLD-Resolution

Input: A first-order program P and a goal list G
Output: An instance $G\delta$ proved from P, or failure
begin
$Res := G$;
While Res is not empty repeat
  Choose a goal g from Res
  Choose a rule $A \leftarrow B_1, \ldots, B_n$, $n \geq 0$, from P such that A and g unify under the mgu δ (renaming variables in the rule as needed);
  If no such rule exists then output failure and exit
  else delete g from Res;
  Add $B_1, \ldots, B_n$ to Res;
  Apply δ to Res and G;
If Res is empty then output $G\delta$
end

57

SLD-Resolution: Example

s(X,Y) ← p(X,Y),q(Y).   q(3).
p(X,3).   q(4).
←s(5,W)

- Goal unifies with rule under {X/5,Y/W}: goal list ← p(5,W),q(W)
- q(W) unifies with q(3) under {W/3}: goal list ← p(5,3)
- Goal unifies with p(X,3) under {X/5}: success with answer {W/3}
- If q(4) is choosen with {W/4}: goal list ← p(5,4)
- Goal cannot unify with head of any rule ⇒ returns failure
- At each step, SLD-resolution choose nondeterministically
  - a next goal from the goal list
  - a next rule from those whose head unifies with goal just selected
- An instance of an SLD-resolution can return success or failure depending on the choices made

58

SLD-Resolution, cont.

- Success set for predicate q without bound arguments: consider all choices and collect results of successful instances of SLD-resolution
- Union of success sets for all predicates in a program P = least model of P ⇒ equivalence between top-down and bottom-up semantics
- Generation of success set for a predicate (e.g., using breadth-first) too inefficient for most practical applications
- Prolog: depth-first exploration of alternatives, left-to-right order of goals, heads of rules in order of appearence
- Programmer responsible to guide Prolog into successful and efficient searches

59

Infinite Loops

anc(X,Z) ← anc(X,Y),parent(Y,Z).
anc(X,Y) ← parent(X,Y).

- Resolvents of ?anc(marc,mary) with 1st rule
  ?anc(marc,Y_1),parent(Y_1,mary).
  ?anc(marc,Y_2),parent(Y_2,Y_1),parent(Y_1,mary).
  ?anc(marc,Y_3),parent(Y_3,Y_2),parent(Y_2,Y_1),parent(Y_1,mary).
  ...
- Reordering does not ensure safety from infinite loops
  anc(X,Y) ← parent(X,Y).
  anc(X,Z) ← anc(X,Y),anc(Y,Z).
- Produce all ancestor pairs and then enter a perpetual loop: e.g., if there is no parent fact, second rule calls itself infinitely
- Even when rules are properly written, directed cycles in parent (e.g., homonyms, incorrect data) cause infinite loops
- Similar rules compute transitive closure of graphs
- Bottom-up operational semantics more robust

60

Reducing Search Space

- Top-down approach take advantage of constants and constraints

  anc(Old,Young) ← parent(Old,Young).
  anc(Old,Young) ← anc(Old,Mid),parent(Mid,Young).
  grandma(Old,Young) ← parent(Mid,Young),mother(Old,Mid).
  parent(F,Cf) ← father(F,Cf).
  parent(M,Cm) ← mother(M,Cm).
- ?grandma(GM,marc): marc unifies with Young, then with Cf and Cm
- Search in father for 2nd column = marc: efficient if index
- A value, say, tom, passed to Mid and mother(Old,tom) is solved
- If several names found for father, each one passed to goal mother and new answers generated for each new name
- When no more names are found, Cm=marc attempted and second parent rule processed similarly
- Deductive DBs mix bottom-up and top-down techniques for combining their strength

61

Recursive Queries in SQL

- SQL3 standards include support for recursive queries

```
CREATE RECURSIVE VIEW allSubparts(Major,Minor) AS
SELECT PART SUBPART
FROM assembly
UNION
SELECT all.Major assb.SUBPART
FROM allSubparts all, assembly assb
WHERE all.Minor=assb.PART
```

- Exit and recursive selects: before and after the union
- A query on this view is needed to materialize the recursive relation

```
SELECT *
FROM allSubparts
```

62

Recursive Queries in SQL

- WITH construct: another, more direct, way to express recursion
- Example: find the parts using 'topTube'

```
WITH RECURSIVE all_super(Major,Minor) AS
    (SELECT PART, SUBPART
    FROM assembly
    UNION
    SELECT assb.PART, all.Minor
    FROM assembly assb, all_super all
    WHERE assb.SUBPART = all.Major
    )
SELECT *
WHERE Minor='topTube'
```

63