Deductive Databases: Topics

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

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

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

<table>
<thead>
<tr>
<th>student</th>
<th>took</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Major</td>
</tr>
<tr>
<td>Joe Doe</td>
<td>cs</td>
</tr>
<tr>
<td>Jim Jones</td>
<td>cs</td>
</tr>
<tr>
<td>Jim Black</td>
<td>ee</td>
</tr>
<tr>
<td>Jim Black</td>
<td>ee</td>
</tr>
</tbody>
</table>

Datalog, cont.

- Fact: logical predicate having only constants (no variables)
- Conventions
  - constants: tokens beginning with lowercase characters or numbers, tokens in quotes
  - variables: tokens beginning with uppercase characters
- Rules: main construct of Datalog programs
  - firstReq(Name) ← student(Name, Major, junior),
    took(Name, cs101, Grade1),
    took(Name, cs123, Grade2).
- Head, body, goals of rules, arity of predicates
- Commas = logical conjuncts, order of goals is immaterial
Datalog, cont.

- Logical disjunct: multiple rules sharing same predicate name and arity
  \[ \text{scndReq}(\text{Name}) \leftarrow \text{student}(\text{Name}, \text{Major}, \text{junior}), \text{took}(\text{Name}, \text{cs131}, \text{Grade}), \text{Grade} > 3.0. \]
  \[ \text{scndReq}(\text{Name}) \leftarrow \text{student}(\text{Name}, \_\text{junior}), \text{took}(\text{Name}, \text{cs151}, \text{Grade}), \text{Grade} > 3.0. \]
- \text{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

<table>
<thead>
<tr>
<th>Datalog</th>
<th>Relational Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base predicate</td>
<td>Table or relation</td>
</tr>
<tr>
<td>Derived predicate</td>
<td>View</td>
</tr>
<tr>
<td>Fact</td>
<td>Row or tuple</td>
</tr>
<tr>
<td>Argument</td>
<td>Columns or attribute</td>
</tr>
</tbody>
</table>

- 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
  \[ \text{req.cs298}(\text{Name}) \leftarrow \text{firstReq}(\text{Name}), \text{scndReq}(\text{Name}). \]

Query Goals

- Specify which of derived relations must be computed
- Boolean or closed queries: contain no variables
- Their answer is yes or no
  \[ ?\text{firstReq}(\text{Jim Black}) \]
- Open queries contain variables
  \[ ?\text{firstReq}(\text{X}) \]
- Their answer is a (possibly empty) set of facts satisfying the query
  \[ \text{firstReq}(\text{Jim Black}) \]
  \[ \text{firstReq}(\text{Jim Black}) \]
- Query goals mix variables and constants in their arguments
Datalog and Negation

- Negation can only be applied to goals of rules.

  \[ \text{hasTaken}(\text{Name}, \text{Course}) \leftarrow \text{took}(\text{Name}, \text{Course}, \text{Grade}). \]

  \[ \text{lacks}_{\text{cs143}}(\text{Name}) \leftarrow \text{student}(\text{Name}, \text{junior}), \neg \text{hasTaken}(\text{Name}, \text{cs143}). \]

- 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.

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.
     \[ \text{reqMissing}(\text{Name}) \leftarrow \text{student}(\text{Name}, \text{senior}), \neg \exists \text{G}1 (\text{took}(\text{Name}, \text{cs101}, \text{G})) \land \exists \text{G}2 (\text{took}(\text{Name}, \text{cs143}, \text{G}2)) \land \exists \text{M} (\text{student}(\text{Name}, \text{M}, \text{junior})). \]
  2. Original query reexpressed as: find senior students who are not missing any requirement for a cs major.
     \[ \text{allReqSat}(\text{Name}) \leftarrow \text{student}(\text{Name}, \text{senior}), \neg \text{reqMissing}(\text{Name}). \]

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 \( \text{?firstReq}(\text{N}) \) in DRC and TRC.

\[
\begin{align*}
\{(\text{N}) | & \exists \text{G}_1 (\text{took}(\text{N}, \text{cs101}, \text{G}_1)) \land \exists \text{G}_2 (\text{took}(\text{N}, \text{cs143}, \text{G}_2)) \land \\
& \exists \text{M} (\text{student}(\text{N}, \text{M}, \text{junior})) \} \\
\{(\text{t}[1]) | & \exists \text{u} (\exists \text{G}(\text{took}(\text{t}[1]) \land \text{took}(\text{u}))) \land \text{student}(\text{s}[3]) \land \text{t}[1] = \text{cs101} \land \\
& \text{t}[2] = \text{cs143} \land \text{t}[1] = \text{u}[1] \land \text{s}[3] = \text{junior} \land \text{s}[1] = \text{t}[1]\}
\end{align*}
\]

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

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 conjunctions and disjunctions in the same formula, negation, explicit existential and universal quantifiers.

- Example: query \( \text{?allReqSat}(\text{N}) \) in DRC.

\[
\begin{align*}
\{(\text{N}) | & \exists \text{M} (\text{student}(\text{N}, \text{M}, \text{junior})) \land \\
& \forall \text{C} (\text{req}(\text{cs}, \text{C}) \rightarrow \exists \text{G} (\text{took}(\text{N}, \text{C}, \text{G}))) \} \\
\end{align*}
\]

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

- **Union**: \( R \cup S = \{ t \mid t \in R \cup t \in S \} \)
- **Difference**: \( R - S = \{ t \mid t \in R \land t \notin S \} \)
- **Cartesian Product**: \( R \times S = \{ t \mid (\exists r \in R)(\exists s \in S) (t[1, \ldots, n] = r \land t[n + 1, \ldots, m] = s) \} \)
- **Projection**: \( \pi_L(R) = \{ r[L] \mid r \in R \}, L \subseteq \{1, \ldots, n\} \)
- **Selection**: \( \sigma_F(R) = \{ t \mid t \in R \land F \} \), \( F \) is composed from \( i \theta C \) and \( i \theta \) using \( \land, \lor, \) and \( \neg \)

Example: \( \sigma_{G>3}(R) = \{ t \mid t \in R \land t[2] > t[3] \} \)

- Relations must be union-compatible

Derived Algebraic Operators

- **Join**: \( R \Join F S = \sigma_{F'}(R \times S) \)
  - \( F = \xi_1 \theta_1 j_1 \ldots \xi_k \theta_k j_k \)
  - \( F' \) denotes the formula obtained from \( F \) by replacing \( \xi_i \theta_j \) by \( t[i] \)
  - and \( t[j] \)
- **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**: \( \pi_L(R) \) is a list of constants and column names (components may appear more than once)
  - \( E \subseteq \pi_{L \times A}(R) \)

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 mapped 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 $\exists q(t_1, \ldots, t_n)$ is safe when predicate $q(A_1, \ldots, A_n)$ is safe

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

   $s(Z,b,W) \leftarrow q(X,Y), p(Y,Z,a), W=Z, W>24.3.$
   
   is translated into

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

2. Body of a rule $r$ translated into RA expression $\text{Body}_r$: Cartesian product of relations in body followed by selection $\sigma_F$ accounting for equalities and inequalities

   $\text{Body}_r = \sigma_{f_1, f_2, \ldots, f_4, f_6, g_5, g_7>24.3}(Q \times P')$

From Safe Datalog to RA, cont.

3. Each rule $r$ is translated into a generalized projection on $\text{Body}_r$, according to the patterns in the head of $r$

   $S = \pi_{f_5, f_5, f_5, f_5}(\text{Body}_r)$

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

Mapping Rules with Negated Goals

$r : \ldots \leftarrow b_1(a,Y), b_2(Y), \neg b_3(Y)$

- Positive body: negated goal is dropped
  $r_P : \ldots \leftarrow b_1(a,Y), b_2(Y)$

- Negative body: remove negation from negated goal
  $r_N : \ldots \leftarrow b_1(a,Y), b_2(Y), b_3(Y)$

- Both bodies are safe, can be transformed into RA expressions giving $\text{Body}_p$ and $\text{Body}_n$

- Body expression to be used in step 3 $\text{Body}_r = \text{Body}_p - \text{Body}_n$

- Rules with several negated goals can be translated by repeating this mapping for each negated goal
Mapping Rules with Several Negated Goals

- Rules with several negated goals can be translated by repeating this mapping for each negated goal, e.g.,
  \[ r : \ldots \rightarrow \neg b_1(X, Y), \neg b_2(X), \neg b_3(Y). \]
- Positive body: all negated goals are dropped
  \[ r_p : \ldots \rightarrow b_1(X, Y). \]
- Negative bodies
  - one for each negated goal
  - add to the positive body one of the negated goals, without negation
  \[ r_m^1 : \ldots \rightarrow b_1(X, Y), b_2(X). \]
  \[ r_m^2 : \ldots \rightarrow b_1(X, Y), b_3(Y). \]
- Body expression: \[ \text{Body}_{r_p} - \text{Body}_{r_m^1} - \ldots - \text{Body}_{r_m^n} \]

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
  \[
  \text{student}(\text{Name}, \text{Yr}) \leftarrow \text{student}(\text{Name}, \text{Cs}, \text{Yr}), \neg \text{took}(\text{Name}, \text{Cs143}, \text{G}).
  \]
  can be viewed as a shorthand to
  \[
  \text{projectTook}(\text{Name}, \text{Cs143}) \leftarrow \text{took}(\text{Name}, \text{Cs143}, \text{G}).
  \]
  \[
  \text{student}(\text{Name}, \text{Yr}) \leftarrow \text{student}(\text{Name}, \text{Cs}, \text{Yr}), \neg \text{projectTook}(\text{Name}, \text{Cs143}).
  \]

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
  \[
  \]
  \[
  \text{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
  \[
  s.Name = t.Name
  \]

Universal Quantification in SQL

- EXISTS and ALL are allowed in nested SQL queries
- Universal quantifiers must be expressed using double negation and existential quantifiers
  \[
  \text{SELECT Name FROM Student WHERE Year='senior' AND Name NOT IN}
  \[
  (\text{SELECT S.Name FROM student s, req r WHERE r.Major='cs' AND s.Year='senior' AND}
  \[
  \not \exists \left( \text{SELECT t.* FROM took t WHERE t.Course=r.Course AND t.Name=s.Name} \right)
  \[
  )
  \]
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 developed 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.

<table>
<thead>
<tr>
<th>partCost</th>
<th>assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicPart</td>
<td>supplier</td>
</tr>
<tr>
<td>topTube</td>
<td>cinelli</td>
</tr>
<tr>
<td>downTube</td>
<td>columbus</td>
</tr>
<tr>
<td>headTube</td>
<td>cinelli</td>
</tr>
</tbody>
</table>

- To find all subparts of a given part a recursive rule is needed.
- 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.
  \[
  \text{basicSubparts}(\text{BasicP, BasicP}) \leftarrow \text{partCost}(\text{BasicP, } \cdot, \cdot),
  \text{basicSubparts}(\text{Part, BasicP}) \leftarrow \text{assembly}(\text{Part, SubP}_\cdot),
  \text{basicSubparts}(\text{SubP, BasicP}).
  \]
- For each basic part, find the least time needed for delivery.

\[
\begin{align*}
  \text{fastest}(\text{Part, Time}) & \leftarrow \text{partCost}(\text{Part, } \cdot, \cdot), \\
  \text{faster}(\text{Part, Time}) & \leftarrow \text{partCost}(\text{Part, Sup}_\cdot, \cdot, \cdot),
  \text{partCost}(\text{Part, Sup}_\cdot, \cdot, \cdot, \cdot),
  \text{Time1} < \text{Time}.
\end{align*}
\]

- Times required for basic subparts of the given assembly.

\[
\begin{align*}
  \text{timeForBasic}(\text{AssPart, BasicSub, Time}) & \leftarrow \\
  \text{basicSubparts}(\text{AssPart, BasicSub}), \text{fastest}(\text{BasicSub, Time}).
\end{align*}
\]

Example: Basic Subparts, cont.

- Maximum time required for basic subparts of the given assembly.

\[
\begin{align*}
  \text{howSoon}(\text{AssPart, Time}) & \leftarrow \text{timeForBasic}(\text{AssPart, _}), \\
  \text{larger}(\text{AssPart, Time}) & \leftarrow \text{timeForBasic}(\text{Part, _}),
  \text{timeForBasic}(\text{Part, _Time1}), \text{Time1} > \text{Time}.
\end{align*}
\]

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

- 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 $g → h$ 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 stratifiable 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.

Functors

- Store complex terms and variable-length subrecords in tuples:
  \[
  \begin{align*}
  & \text{part}(202, \text{circle}(11), \text{actualKg}(0.034)). \\
  & \text{part}(21, \text{rectangle}(10, 20), \text{unitKg}(2.1)). \\
  & \text{partWeight}(\text{No}, \text{Kilos}) \leftarrow \text{part}(\text{No}, \text{actualKg}(\text{Kilos})). \\
  & \text{partWeight}(\text{No}, \text{Kilos}) \leftarrow \text{part}(\text{No}, \text{unitKg}(\text{Kilos})). \\
  & \text{area}((\text{Shape}, A) \leftarrow A = \text{Dmtr}^2 \times 3.14 / 4. \\
  & \text{area}((\text{Base}, \text{Height}), A) \leftarrow A = \text{Base} \times \text{Height}.
  \end{align*}
\]
- In actual applications, functions are used as variable-length subrecords.
- Functions can be nested, can be used as discriminants to prescribe different computations.

Lists

- Functions 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:
  \[
  \text{partSupList(topTube, [cinelli, columbus, mavic]).}
  \]
- Normalizing a nested relation into a flat relation:
  \[
  \begin{align*}
  & \text{flatten}(\text{P}, S, L) \leftarrow \text{partSupList}(\text{P}, S \mid L). \\
  & \text{flatten}(\text{P}, S, L) \leftarrow \text{flatten}(\text{P}, S \mid L). \\
  & \text{ps}(\text{Part}, \text{Sup}) \leftarrow \text{flatten}(\text{Part}, \text{Sup}).
  \end{align*}
  \]
- Applying these rules yield:
  \[
  \begin{align*}
  & \text{ps(topTube, cinelli).} \\
  & \text{ps(topTube, columbus).} \\
  & \text{ps(topTube, mavic).}
  \end{align*}
  \]

Lists, cont.

- Constructing a nested relation from a normalized relation:
  \[
  \begin{align*}
  & \text{between}(\text{P}, \text{X}, \text{Z}) \leftarrow \text{ps}(\text{P}, \text{X}), \text{ps}(\text{P}, \text{Y}), \text{ps}(\text{P}, \text{Z}), \text{X} < \text{Y}, \text{Y} < \text{Z}. \\
  & \text{smaller}(\text{P}, \text{X}) \leftarrow \text{ps}(\text{P}, \text{X}), \text{ps}(\text{P}, \text{Y}), \text{Y} < \text{X}. \\
  & \text{nested}(\text{P}, \text{X}) \leftarrow \text{ps}(\text{P}, \text{X}), \text{ps}(\text{P}, \text{X}). \\
  & \text{nested}(\text{P}, \text{Y}[\text{X}], \text{W}) \leftarrow \text{nested}(\text{P}, \text{W}), \text{ps}(\text{P}, \text{Y}), \text{X} < \text{Y}, \text{--between}(\text{P}, \text{X}, \text{Y}). \\
  & \text{psNest}(\text{P}, \text{W}) \leftarrow \text{nested}(\text{P}, \text{W}), \text{--nested}(\text{P}, \text{X}).
  \end{align*}
  \]
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: \( \lor, \land, \neg, \leftarrow, \rightarrow, \leftrightarrow \)
- 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 \( \neg F, F \lor G, F \land G, F \leftarrow G, F \rightarrow G \).
  - 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
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

\[
\begin{align*}
\text{anc}(X,Y) & \leftarrow \text{parent}(X,Y). \\
\text{anc}(X,Z) & \leftarrow \text{anc}(X,Y), \text{parent}(Y,Z). \\
\text{parent}(X,Y) & \leftarrow \text{father}(X,Y). \\
\text{parent}(X,Y) & \leftarrow \text{mother}(X,Y). \\
\text{mother}(\text{anne}, \text{silvia}). \\
\text{mother}(\text{anne}, \text{marc}).
\end{align*}
\]

- $U_P = \{\text{anne, silvia, marc}\}
- B_P = \{[\text{parent}(x, y) | x, y \in U_P] \cup [\text{father}(x, y) | x, y \in U_P] \cup [\text{mother}(x, y) | x, y \in U_P] \cup [\text{anc}(x, y) | x, y \in U_P]
- |B_P| = 4 \times 3 \times 3 = 36
- 2^{|B_P|} = 2^{36}$ Herbrand interpretations
Herbrand Interpretations, cont.

- Program $P$ with an infinite $B_P$ and an infinite number of interpretations
  - $p(f(X)) \leftarrow q(X)$.
  - $q(a) \leftarrow p(X)$.

- $U_P = \{a, f(a), \ldots, f^n(a), \ldots\}$, 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

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

- Ground version of a program $P$:
  - $\text{ground}(P) = \{\text{ground}(r) \mid r \in P\}$

- 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 $\text{ground}(P)$

- Interpretations and models for the example
  - $I_1 = \emptyset$ is not a model: facts are not satisfied
  - $I_2 = \{\text{mother}(\text{anne}, \text{silvia}), \text{mother}(\text{anne}, \text{marc})\}$ is not a model
  - $I_3 = \{\text{mother}(a,s), \text{mother}(a,m), \text{parent}(a,s), \text{parent}(a,m), \text{anc}(a,s), \text{anc}(a,m)\}$ is a model
  - $I_4 = I_3 \cup \{\text{anc}(\text{silvia}, \text{marc})\}$ 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 $M' \subseteq M$
- 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 argument and the result of the RA expression
- Fixpoint equation: \( x = T(x) \) where \( T \) is a mapping \( U \rightarrow U \)
- Fixpoint for \( T \): a value \( x \) that satisfies this equation
- For an arbitrary \( T \) there might be zero or more fixpoints.

Immediate Consequence Operator

- Mapping \( T_P \), the immediate consequence operator for \( P \)
  \[ T_P = \{ A \in B_P \mid \exists r: A \leftarrow A_1, \ldots, A_n \in \text{ground}(P), \{A_1, \ldots, A_n\} \subseteq I \} \]
- Is a mapping from HIs of \( P \) to HIs of \( P \)
- For the program \( P \)
  - \( \text{anc}(X,Y) \leftarrow \text{parent}(X,Y) \)
  - \( \text{anc}(X,Z) \leftarrow \text{anc}(X,Y), \text{parent}(Y,Z) \)
  - \( \text{parent}(X,Y) \leftarrow \text{father}(X,Y) \)
  - \( \text{parent}(X,Y) \leftarrow \text{mother}(X,Y) \)
  - \( \text{mother}(\text{anne}, \text{silvia}) \), \( \text{mother}(\text{anne}, \text{marc}) \).
- \( I = \{ \text{anc}(\text{anne}, \text{marc}), \text{parent}(\text{marc}, \text{silvia}) \} \)
- \( T_P = \{ \text{anc}(\text{marc}, \text{silvia}), \text{anc}(\text{anne}, \text{silvia}), \text{mother}(\text{anne}, \text{silvia}), \text{mother}(\text{anne}, \text{marc}) \} \)
- In addition to the atoms derived from the applicable rules, \( T_P \) returns also the facts and the ground instances of unit clauses.

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 \( \subseteq \).
  - 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 if it contains infinitely many elements.
- \( T_P \) for definite clause programs is monotonic, i.e., if \( N \subseteq M \) then \( T_P(N) \subseteq T_P(M) \).
- If \( P \) is a definite clause program, then
  - there always exists a least fixpoint for \( T_P \), denoted \( \text{lfp}(T_P) \)
  - \( M_P = I \cap \text{lfp}(T_P) \).
Arguments can be complex

• 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

• For positive programs, \( T_P \) computed by repeated applications of \( T_P \)

\[ T_P^0(I) = I \]

\[ T_P^{n+1}(I) = T_P(T_P^n(I)) \]

• With \( \omega \) denoting the first limit ordinal

\[ T_P^n(I) = \bigcup \{ T_P^n(I) \mid n \geq 0 \} \]

• For a definite clause program \( P \), \( T_P(P) = T_P^\omega(\emptyset) \)

• This gives a simple algorithm for computing \( T_P(P) \)

  – 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

\[
\begin{align*}
\text{part}(202,\text{circle}(11),\text{actualKg}(0.034)). \\
\text{part}(21,\text{rectangle}(10.20),\text{actualKg}(2.1)). \\
\text{partWeight}(\text{No}, \text{Kilos}) \leftarrow \text{part}(\text{No}, \text{actualKg}(\text{Kilos})). \\
\text{partWeight}(\text{No}, \text{Kilos}) \leftarrow \text{part}(\text{No}, \text{Shape}, \text{unitKg}(\text{Kilos})). \\
\text{area}(\text{Shape}, \text{Area}). \text{Kilos} \leftarrow \text{Kilos} \times \text{area}. \\
\text{area}(\text{circle}(\text{Dmtr}), A) \leftarrow \text{A=} \text{Dmtr} \times \text{Dmtr}^2 / 4. \\
\text{area}(\text{rectangle}(\text{Base}, \text{Height}), A) \leftarrow \text{A=} \text{Base} \times \text{Height}.
\end{align*}
\]

• \( A \) and \( \text{Area} \approx \) formal and actual parameters in procedural languages

• Arguments can be complex \( \Rightarrow \) passing of parameters through unification

---

Powers of \( T_P \)

\[
\begin{align*}
\text{For positive programs, } T_P & \text{ computed by repeated applications of } T_P \\
T_P^n(I) & = I \\
T_P^{n+1}(I) & = T_P(T_P^n(I)) \\
\text{With } \omega \text{ denoting the first limit ordinal} \\
T_P^n(I) & = \bigcup \{ T_P^n(I) \mid n \geq 0 \} \\
\text{For a definite clause program } P, T_P & = T_P^\omega(\emptyset) \\
\text{This gives a simple algorithm for computing } T_P(P) \\
\text{– Starting from the bottom} \\
\text{– Iterating the application of } T \text{ ad infinitum or until no new atoms are} \\
\text{obtained and the } (n+1)^{th} \text{ power equals the } n^{th} \text{ power}
\end{align*}
\]

---

Unification

• Substitution \( \theta \): \(\{v_1/t_1, \ldots, v_n/t_n\} \), 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 \( \theta \) a substitution for variables in \( E \), then \( E\theta \) is the result

of applying \( \theta \) to \( E \)

\[ \text{E.g., } E = p(x, y, f(a)); \theta = \{x/b, y/x\} \Rightarrow E\theta = p(b, x, f(a)) \]

• Variables that are not part of the substitution left unchanged

• Composition \( \delta \theta \) 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 \( \{u_i/s_i\delta, \ldots, u_m/s_m\delta, v_1/t_1, \ldots, v_n/t_n\} \) by deleting

  any \( u_i/s_i\delta \) for which \( u_i = s_i\delta \) and deleting any \( v_j/t_j \) for which

  \( v_j \in \{u_1, \ldots, u_m\} \)

\[ \text{E.g., } \theta = \{x/f(y), y/z\}, \delta = \{x/a, y/b, z/y\}; \delta \theta = \{x/f(b), z/y\} \]

---

Unifiers

• Substitution \( \theta \) is a unifier for terms \( A \) and \( B \) if \( A\theta = B\theta \)

• Most general unifier (mgu): unifier for two terms such that for each other

unifier \( \gamma \) there is a substitution \( \delta \) such that \( \gamma = \delta \theta \)

\[ \text{E.g., } p(f(x), a) \text{ and } p(g, f(u)) \text{ not unifiable} \]

• 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 \( g \leftarrow r \) (\( r \) and \( g \) have no variables in common),

if there is an mgu \( \delta \) for \( A \) and \( g \), the resolvent of \( r \) and \( g \leftarrow r \) 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 $\delta$ (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 $\delta$ to Res and $G$;
If Res is empty then output $G\delta$
end

SLD-Resolution: Example

\[ s(X,Y) \leftarrow p(X,Y), q(Y). \]
\[ q(3). \]
\[ p(X,3). q(4). \]
\[ \leftarrow s(5,W) \]
• Goal unifies with rule under \{X/5,Y/W\}
• goal list $\leftarrow p(5,W), q(W)$
• $q(W)$ unifies with $q(3)$ under \{W/3\}
• goal list $\leftarrow p(5,3)$
• Goal unifies with $p(X,3)$ under \{X/5\}: success with answer \{W/3\}
• If $q(4)$ is chosen with \{W/4\}: goal list $\leftarrow p(5,4)$
• Goal cannot unify with head of any rule $\Rightarrow$ 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

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 $P$ $\Rightarrow$ 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 appearance
• Programmer responsible to guide Prolog into successful and efficient searches

Infinite Loops

\[ \text{anc}(X,Z) \leftarrow \text{anc}(X,Y), \text{parent}(Y,Z). \]
\[ \text{anc}(X,Y) \leftarrow \text{parent}(X,Y). \]
• Resolvents of $\text{anc}(\text{marc}, \text{mary})$ with 1st rule
  $\text{anc}(\text{marc}, Y_1). \text{parent}(Y_1, \text{mary})$. $\text{anc}(\text{marc}, Y_2). \text{parent}(Y_2, Y_1). \text{parent}(Y_1, \text{mary})$. $\text{anc}(\text{marc}, Y_2). \text{parent}(Y_2, Y_1). \text{parent}(Y_1, Y_2). \text{parent}(Y_2, \text{mary}).$
• Reordering does not ensure safety from infinite loops
  $\text{anc}(X,Y) \leftarrow \text{parent}(X,Y).$ \text{anc}(XZ) $\leftarrow \text{anc}(X,Y). \text{anc}(Y,Z).$
• 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
Reducing Search Space

- Top-down approach take advantage of constants and constraints
  \[ \text{anc}(\text{Old}, \text{Young}) \leftarrow \text{parent}(\text{Old}, \text{Young}). \]
  \[ \text{anc}(\text{Old}, \text{Young}) \leftarrow \text{anc}(\text{Old}, \text{Mid}), \text{parent}(\text{Mid}, \text{Young}). \]
  \[ \text{grandma}(\text{Old}, \text{Young}) \leftarrow \text{parent}(\text{Mid}, \text{Young}), \text{mother}(\text{Old}, \text{Mid}). \]
  \[ \text{parent}(\text{F}, \text{Cf}) \leftarrow \text{father}(\text{F}, \text{Cf}). \]
  \[ \text{parent}(\text{M}, \text{Cm}) \leftarrow \text{mother}(\text{M}, \text{Cm}). \]
- \(?\text{grandma}(\text{GM}, \text{marc})\):
  - marc unifies with \(\text{Young}\), then with \(\text{Cf}\) and \(\text{Cm}\)
  - Search in \(\text{father}\) for 2nd column = marc: efficient if index
  - A value, say, \(\text{tom}\), passed to \(\text{Mid}\) and \(\text{mother}(\text{Old}, \text{tom})\) is solved
  - If several names found for \(\text{father}\), each one passed to \(\text{goal}\) mother
  - When no more names are found, \(\text{Cm} = \text{marc}\) attempted and second \(\text{parent}\) rule processed similarly
- Deductive DBs mix bottom-up and top-down techniques for combining their strength

Recursive Queries in SQL

- SQL3 standards include support for recursive queries
  
  ```sql
  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
  ```sql
  SELECT *
  FROM allSubparts
  ```