Datalog

A language of **logical facts and rules**.

A subset of Prolog (no data structures).

Basic concepts:

- predicate (relation)
- term, constant, variable
- goal
- clause, rule, fact
- substitution
- unification

Terms

Terms can be:

- constant symbols:
  - *atoms* (names)
  - *numbers* (integers, floats)
- (logical) variable symbols

Variables and constants are distinguished using a syntactic convention (e.g., variables start with an upper-case letter).

A logical **variable**:

- placeholder (stands for any value)
- can be **instantiated** by substituting it by another term
- cannot be assigned (no assignment in the language).

Goals and clauses

Concept	*Syntax*	Semantics
Goal	$P(T_1, \ldots, T_n)$	Predicate P is true of terms T_1 and ... and T_n
Query	$A_1, \ldots, A_k$	A_1 and ... A_k are true
Implication	$A_0 \ :- \ A_1, \ldots, A_k$	A_0 if A_1 and $\cdots$ and A_k

The *head* of an implication is A_0, the *body* is $A_1, A_2, \ldots, A_k$. The body may be true ($k = 0$) and ": $-$" is then skipped. No variables appear only in the head.

Recursive rule: the same predicate appears in the head and the body of a rule.

An implication is also called a **clause**:

- *unit clause* (fact) - if empty body.
- *nonunit clause* (rule) - if nonempty body.

Typically, predicates are defined only using facts (**EDB** predicates) or only rules (**IDB** predicates).

A **logic program** is a collection of clauses viewed logically as their conjunction.

Example

Program P_1: determining ancestors.

```
%% Rules
anc(X,Y) :- parent(X,Y).
anc(X,Z) :- parent(X,Y), anc(Y,Z).

%% Facts
parent(witold,tom).
parent(tom,jan).
parent(tom,tony).
parent(jan,dave).

%% Query 1
anc(X,dave)

%% Query 2
anc(X,dave), anc(X,tony)
```

Another example

Public-key infrastructure:

- principals: (public key,Internet address)

pkd(P,N,K) $\equiv$ the principal P has the principal key K for the name N.

```
pkd(p_alice,alice,p_alice).
pkd(p_alice,bob,p_bob).
pkd(p_alice,X,Y) :- pkd(p_alice,bob,Z), pkd(Z,X,Y).
```

Can also be used to model PGP, DNS, LDAP,...

Logical meaning of a logic program

The least Herbrand model M_P:

- the set of facts implied by the program
- contains all the facts in the program
- contains also all the facts that can be derived (directly or indirectly) by the rules
- contains no other facts

M_{P_1} consists of all `parent` facts and:

```
%% First-level ancestors
anc(witold,tom).
anc(tom,jan).
anc(tom,tony).
anc(jan,dave).

%% Second-level ancestors
anc(witold,jan).
anc(witold,tony).
anc(tom,dave).

%% Third-level ancestors
anc(witold,dave).
```

Substitutions

In logic programs, variables are implicitly **universally quantified**. This means that every implication is true for all instantiations to its variables (the occurrences of the same variable in a single clause have to be instantiated in the same way).

Formally, a **substitution** is a mapping of variables to terms. A substitution h can be applied to a goal G by replacing all the variables in the domain of h in such a way that all the occurrences of a variable x are replaced by the same term $h(x)$. This yields the goal Gh (an instance of G).

Substitutions can be applied to clauses and **composed**. A **ground** substitution replaces all the variables in its domain by constants. The **ground version** $ground(P)$ of a program P consists of all the ground instances of the clauses in P.

Fixpoint semantics

For a given logic program P and a set of ground facts I:

$$\begin{aligned} T_P(I) \quad = \quad & \{A | \exists r \in ground(P). \\ & r = A :- A_1, \ldots, A_n \wedge A_1 \in I \wedge \cdots A_n \in I\} \end{aligned}$$

I is a **fixpoint** of T_P if $T_P(I) = I$.

Properties of T_P:

1. T_P has a least fixpoint $lfp(T_P)$ (contained in all the fixpoints)
2. $lfp(T_P)$ is obtained by iterating T_P finitely many times
3. $lfp(T_P) = M_P$

Datalog: query evaluation

Bottom-up:

- compute

  $$T_P(\emptyset), T_P(T_P(\emptyset)), \ldots$$

  until the result does not change
- at the end evaluate the query using the obtained set of facts (M_P)
- find appropriate substitutions by matching the goals in the body against the set of facts
- built-in predicates ($=, <, \ldots$) should be made true by the substitutions
- the evaluation terminates and can be further optimized to avoid duplicating work and computing unnecessary facts (by using information in the query).

Basic mode for most deductive DBMS: LDL (LDL++), Coral, ...

Top-down query evaluation

Evaluation of a ground goal A:

1. A **succeeds** immediately if there is a fact A in the program
2. A **succeeds** if there is a clause $A_0 : -A_1, \ldots, A_n$ in the program and a ground substitution h such that
   - $h(A_0) = A$
   - the evaluation of $h(A_1)$, $h(A_2)$, ..., $h(A_n)$ all succeed

Failure: backtracking (substitutions are undone).

Infinite looping: tabling (some systems, e.g. XSB).

Finding substitutions

Evaluation of a non-ground goal A:

- **unify** A with a fact or a rule head (after renaming apart)
- **propagate** the substitutions to the body of the rule
- **evaluate** the body

Two goals **unify** if there is a substitution that maps both to the same goal. A substitution is a **most general unifier** (mgu) of two goals if all other unifying substitutions can be obtained from it by composition.

Unification algorithm

Unification of two goals `G1` and `G2` that either **fails**, or **succeeds** and returns an **mgu** `h`:

```
h:={}
if predicate(G1)<>predicate(G2) or arity(G1)<>arity(G2)
   then success  := false
else for i=1 to arity(G1) do
       A1=arg(G1,i); A2=arg(G2,i)
       if variable(A1) and A1<>A2 then
          instantiate all the occurrences of A1
          in G1 and h to A2
          add (A1,A2) to h
       else if variable(A2) and A1<>A2 then
          instantiate all the occurrences of A2
          in G2 and h to A1
          add (A2,A1) to h
       else if A1<>A2 then
          success  := false
success := true
```

Open vs. Closed World Assumption

Closed World Assumption (**CWA**:

What is not implied by a program is **false**.

Open World Assumption (**OWA**):

What is not implied by a program is **unknown**.

Scope:

- traditional database applications: CWA
- information integration: OWA or CWA

Datalog ¬

Rules with negated goals in the body:

$$A_0 :- A_1, \ldots, A_k, not\ B_1, \ldots, not\ B_m.$$

Example:

```
forebear(X,Y) :- anc(X,Y), not parent(X,Y).
```

Problems with negation:

- not every program has a clear logical meaning (due to the interaction of negation with recursion)
- bottom-up evaluation does not always produce an intuitive result

Example:

```
p(X) :- not q(X).
q(X) :- not p(X).
```

Implicit quantification

Example:

```
bachelor(X) :- male(X), not married(X,Y).
```

Does it mean:

1. `X` is a bachelor if `X` is a male and `X` is not married to **anyone**, or
2. `X` is a bachelor if `X` is a male and `X` is not married to **everyone**?

Logically:

1. $(\forall X)(bachelor(X) \leftarrow male(X) \wedge (\neg\exists Y)(married(X,Y)))$
2. $(\forall X)(bachelor(X) \leftarrow male(X) \wedge (\exists Y)(\neg married(X,Y)))$

The proper logical reading is 2, because it is equivalent to:

$$(\forall X)(\forall Y)(bachelor(X) \leftarrow male(X) \wedge (\neg married(X, Y)))$$

If the reading 1 is desired, replace the rule by:

```
bachelor(X) :- male(X), not husband(X).
husband(X) :- married(X,Y).
```

We will assume that every variable that appears in a negative goal appears also in a positive goal.

Stratified programs

The **dependency graph** $pdg(P)$ of a logic program P:

- vertices: predicates of P
- edges:
  - a **positive** edge (p, q) if there is a clause in P in which p appears in a positive goal in the body and q appears in the head
  - a **negative** edge (p, q) if there is a clause in P in which p appears in a negative goal in the body and q appears in the head

A Datalog¬ program P is **stratified** if no cycle in its dependency graph $pdg(P)$ contains a negative edge.

Stratifications

Stratification of P: a mapping s from the set of predicates in P to nonnegative integers such that:

1. if a positive edge (p, q) is in $pdg(P)$, then $s(p) \leq s(q)$
2. if a negative edge (p, q) is in $pdg(P)$, then $s(p) < s(q)$

There is a **polynomial-time** algorithm to:

- determine whether a program is stratified,
- if it is, to find a stratification for it

Datalog¬: query evaluation

Bottom-up:

1. compute a stratification of a program P
2. partition P into $P_1, \ldots, P_n$, each P_i consisting of all and only rules whose head belongs to a single stratum
3. evaluate bottom-up $P_1, \ldots, P_n$ (in that order):
   - find the substitutions to the positive goals first
   - use negative subgoals only as tests
   - $\neg A$ succeeds if A is not in the result of the lower strata

The result of bottom-up evaluation:

- does not depend on the stratification
- can be semantically characterized in various ways

Expressive power

There are some queries that are not expressible in relational algebra but expressible in Datalog:

- **transitive closure** of a binary relation

There are some queries that are not expressible in Datalog but expressible in relational algebra:

- set difference (**nonmonotonic** query)

Every relational algebra query can be expressed in Datalog$^{\neg}$.

Computational complexity

Data complexity: complexity as a function of the number of the **facts** (tuples) in the logic program (database).

Bottom-up computation for Datalog and Datalog¬ terminates in **polynomial time**.

Recursion in SQL3

General form:

`WITH` R AS *definition of* R *query to* R

If R is recursively defined, it should be preceded by `RECURSIVE`.

Example:

```
WITH RECURSIVE Anc(Upper,Lower) AS
     (SELECT * FROM Parent)
  UNION
     (SELECT P.Upper, A.Lower
      FROM Parent AS P, Anc AS A
      WHERE P.Lower=A.Upper)
SELECT Anc.Upper
FROM Anc
WHERE Anc.Lower='Dave';
```

Recursion in SQL3 (cont'd)

Mutual recursion: more than relation can be defined simultaneously.

Linear recursion: each definition can have only one occurrence of a relation mutually recursive with the relation being defined.

Negation and recursion: if `EXCEPT` is used, the definitions should be stratified.

Disjunction

Disjunctive facts or rule heads:

$$A_1(\ldots) \textbf{ or } \ldots \textbf{ or } A_k(\ldots).$$
$$A_1(\ldots) \textbf{ or } \ldots \textbf{ or } A_k(\ldots) :- B_1(\ldots), \ldots, B_m(\ldots).$$

Examples:

```
student(123456,toronto) or student(123456,buffalo).

bloodtype(Z,T1) or bloodtype(Z,T2) :-
  parent(X,Z), parent(Y,Z), X\==Y,
  bloodtype(X,T1), bloodtype(Y,T2).
```

Properties:

- a logic program may have multiple **minimal** models, not necessarily the (single) least model
- Closed-World Assumption is typically inconsistent and needs to be modified
- disjunctive logic programs are computationally harder than those without disjunction.

Incomplete information

Different kinds of incompleteness:

- the object is an element of a finite set (but we don't know which one)

  $\Rightarrow$ disjunctive logic programs

- the object exists but we don't know anything about it

  $\Rightarrow$ SQL null

- ...

Constraint databases

Finite representation of infinite, multidimensional sets using **constraints**.

Constraint theory:

- a domain: reals, integers, uninterpreted constants...
- atomic formulas

Atomic constraints:

- should be closed under negation and quantifier elimination
- equalities $x = y$ and disequalities $x \neq y$
- order constraints $x < y$ and $x \leq y$
- linear arithmetic constraints $a_1x_1 + \cdots a_kx_k < a_0$ and $a_1x_1 + \cdots a_kx_k \leq a_0$.
- ...

Constraint databases: basic notions

Generalized tuple: conjunction of atomic constraints. Semantics: the set of tuples satisfying the formula.

Generalized relation: finite set of generalized tuples. Semantics: the set of tuples satisfying at least one generalized tuple in this set.

Example:

$$X > 2 \wedge Y > 2 \wedge X > Y$$

$$X > 0 \wedge X < 1 \wedge Y > 0 \wedge Y < 1$$

Relational algebra for constraint databases

Selection $\sigma_C(R)$: conjoin C with every generalized tuple in R.

Cartesian product $R \times S$: take $t \wedge s$ for every $t \in R$ and $s \in S$.

Union $R \cup S$: union of sets of generalized tuples R and S.

Difference $R - S$:

1. write down the complement of S as a formula
2. put it in disjunctive normal form
3. conjoin every resulting generalized tuple with every tuple in S in turn.

Projection $\pi_X(R)$: apply quantifier elimination to every generalized tuple in R in turn.

Fourier elimination

Quantifier elimination procedure for linear arithmetic constraints and order constraints over a dense domain (rational or reals).

A conjunction of linear arithmetic constraints

$$a_1 x_1 + \cdots + a_n x_n + y \leq a_0$$
$$b_1 x_1 + \cdots + b_n x_n - y \leq b_0$$

can be replaced by a single arithmetic constraint

$$(a_1 + b_1)x_1 + \cdots (a_n + b_n)x_n \leq a_0 + b_0$$

from which y has beeen eliminated (projected out).

This is repeated for every pair of constraints in a generalized tuple. The resulting constraints are conjoined together.

Datalog for constraint databases

Facts defined by generalized tuples.

Termination:

- equalities/disequalities, order constraints: can be obtained
- linear arithmetic constraints: cannot be obtained.