A Brief Introduction to Deductive Databases

Michael Genesereth
CodeX: The Stanford Center for Legal Informatics
Stanford University
### Deductive Database Example

**Facts:**

- `parent(art,bob)`
- `parent(art,bea)`
- `parent(bob,cal)`
- `parent(bob,coe)`

**Rules:**

- `grandparent(X,Z) :- parent(X,Y) & parent(Y,Z)`
- `sibling(Y,Z) :- parent(X,Y) & parent(X,Z) & Y!=Z`

**Conclusions:**

- `grandparent(art,cal)`
- `sibling(bob,bea)`
- `grandparent(art,coe)`
- `sibling(bea,bob)`
- `sibling(cal,coe)`
- `sibling(coe,cal)`

---

A deductive database is a finite collection of facts and rules. By applying the rules of a deductive database to the facts in the database, it is possible to infer additional facts, i.e. facts that are implicitly true but are not explicitly represented in the database.
Facts
The vocabulary of a database is a collection of object constants, function constants, and relation constants. Each function constant and relation constant has an associated arity, i.e. the number of objects involved in any instance of the corresponding function or relation.
A term is either an object constant or a functional term. A functional term is an expression formed from an $n$-ary function constant and $n$ terms enclosed in parentheses and separated by commas.

\[
\begin{align*}
\text{pair}(\text{art}, \text{art}) \\
\text{pair}(\text{art}, \text{bob}) \\
\text{pair}(\text{bob}, \text{art})
\end{align*}
\]

Functional terms are terms and so can be nested.

\[
\text{pair}(\text{pair}(\text{art}, \text{bob}), \text{pair}(\text{bob}, \text{art}))
\]
A datum / fact is an expression formed from an n-ary relation constant and n terms enclosed in parentheses and separated by commas.

\[ \text{parent(art,bob)} \]

A datum / fact is an expression formed from an n-ary relation constant and n terms. For example, if parent is a relation constant with arity 2 and if art and bob are terms, then the expression shown here is a syntactically legal relational sentence.
A *dataset* is a set of data/facts.

<table>
<thead>
<tr>
<th>parent(art,bob)</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent(art,bea)</td>
</tr>
<tr>
<td>parent(bob,chris)</td>
</tr>
<tr>
<td>parent(bob,coe)</td>
</tr>
<tr>
<td>parent(coe,daniel)</td>
</tr>
<tr>
<td>parent(coe,daisy)</td>
</tr>
<tr>
<td>adult(art)</td>
</tr>
<tr>
<td>male(art)</td>
</tr>
<tr>
<td>female(bea)</td>
</tr>
<tr>
<td>adult(bob)</td>
</tr>
<tr>
<td>male(bob)</td>
</tr>
<tr>
<td>female(coe)</td>
</tr>
<tr>
<td>adult(bea)</td>
</tr>
<tr>
<td>male(chris)</td>
</tr>
<tr>
<td>female(daisy)</td>
</tr>
<tr>
<td>adult(chris)</td>
</tr>
<tr>
<td>male(daniel)</td>
</tr>
<tr>
<td>adult(coe)</td>
</tr>
<tr>
<td>prefers(art,bob,bea)</td>
</tr>
<tr>
<td>prefers(coe,daisy,daniel)</td>
</tr>
</tbody>
</table>

A dataset is any set of data that can be formed from the vocabulary of a database. Intuitively, we can think of the data in a dataset as the facts that we believe to be true in the world; data that are not in the dataset are assumed to be false.

As an example of these concepts, consider a small interpersonal database. The objects in this case are people. The relationships specify properties of these people and their interrelationships.
The order of arguments in an instance of a relation is determined by one’s understanding of the relation.

Example:

\texttt{prefers(art,bob,bea)}

For me, this sentence means that Art prefers Bea to Bob. Other interpretations are possible; the important thing is to be consistent - once you choose, stick with it.

Note on Order of Arguments

The order of arguments in such sentences is arbitrary. Given the meaning of the \texttt{prefers} relation in our example, the first argument denotes the subject, the second argument is the person who is preferred, and the third argument denotes the person who is less preferred. We could equally well have interpreted the arguments in other orders. The important thing is consistency – once we choose to interpret the arguments in one way, we must stick to that interpretation everywhere.
Rules
Logic programs are built up from four disjoint classes of symbols, viz. object constants, function constants, relation constants, and variables. In what follows, we write such symbols as strings of letters, digits, and a few non-alphanumeric characters (e.g. "_""). Constants must begin with a lower case letter or digit. Variables must begin with an upper case letter. The arity of a function constant or relation constant is the number of “arguments” that can be “associated” with the function or relation when writing expressions (as we shall see).
Terms:

Object Constants: art, bob

Variables: x, y, z

Functional Terms: pair(x, bob)

A term is either an object constant, a variable, or a functional term. A functional term is an expression consisting of an n-ary function constant and n simpler terms. In what follows, we write functional terms in traditional mathematical notation - the function constant followed by its arguments enclosed in parentheses and separated by commas.
<table>
<thead>
<tr>
<th>Atoms</th>
<th>Negations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(a, b) ), ( p(a, f(X)) ), ( p(g(X, Y), c) )</td>
<td>( \neg p(X, b) )</td>
</tr>
</tbody>
</table>

A literal is either an atom or a negation of an atom.

\( p(a, Y), \neg p(a, Y) \)

An atom is a positive literal.
A negations is a negative literal.

An atom is an expression consisting of an \( n \)-ary relation constant and \( n \) terms. In what follows, we write atoms in traditional mathematical notation - the relation constant followed by its arguments enclosed in parentheses and separated by commas. For example, if \( p \) is a binary relation constant and if \( a \) and \( b \) are object constants, then \( p(a, b) \) is an atom. A negation is an expression formed using the negation sign \( \neg \) and an atom. For example, \( \neg p(a, b) \). A literal is either an atom or a negation. An atom is sometimes called a positive literal, and a negation is sometimes called a negative literal.
A rule is an expression consisting of a distinguished atom, called the head, and a conjunction of zero or more literals, called the body, separated by the :- operator. The literals in the body are called subgoals.

The intended meaning is that an instance of the head is true whenever corresponding instances of all of the positive subgoals are true and all of the negative subgoals are false.
A logic program is a finite set of atoms and rules.

There are a few restrictions, which we discuss later. But, first, some examples.

A logic program is a finite set of atoms and rules of this form. In order to simplify our definitions and analysis, we occasionally talk about infinite sets of rules. While these sets are useful, they are not themselves logic programs. Here are some examples. In this first example, we define the parent relation $p$ in terms of father $f$ and mother $m$. In the second example, we define grandparent in terms of parent. In the third, we define ancestor in terms of parent. Note that the definition in this case is recursive. Finally, we define remote ancestor as any ancestor that is not a parent.

Although every logic program is a finite set of atoms and rules, not every finite set of atoms and rules is a logic program. As mentioned earlier, there are some restrictions to ensure that such descriptions have desirable properties.
The principle use of rules is to define new relations in terms of existing relations. The new relations defined in this way are often called "view relations" (or simply views) to distinguish them from "base relations", which are defined by explicit enumeration of instances.
Fathers and Mothers:

\[
\begin{align*}
\text{father}(X,Y) & :\ - \text{parent}(X,Y) \ & \text{male}(X) \\
\text{mother}(X,Y) & :\ - \text{parent}(X,Y) \ & \text{female}(X)
\end{align*}
\]

<table>
<thead>
<tr>
<th>Data</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent(art,bob)</td>
<td>father(art,bob)</td>
</tr>
<tr>
<td>parent(art,bea)</td>
<td>father(art,bea)</td>
</tr>
<tr>
<td>parent(bob,chris)</td>
<td>father(bob,chris)</td>
</tr>
<tr>
<td>parent(bob,coe)</td>
<td>father(bob,coe)</td>
</tr>
<tr>
<td>parent(coe,daniel)</td>
<td>mother(coe,daniel)</td>
</tr>
<tr>
<td>parent(coe,daisy)</td>
<td>mother(coe,daisy)</td>
</tr>
</tbody>
</table>

As an example, consider the sentences shown below. The first sentence defines the father relation in terms of parent and male. The second sentence defines mother in terms of parent and female.
The rule here defines the grandparent relation in terms of the parent relation. A person \( X \) is the grandparent of a person \( Z \) if \( X \) is the parent of a person \( Y \) and \( Y \) is the parent of \( Z \). The variable \( Y \) here is a "thread variable" that connects the first subgoal to the second but does not itself appear in the head of the rule.
Personhood:

\[
\text{person}(X) :- \text{male}(X) \\
\text{person}(X) :- \text{female}(X)
\]

Data: 
- male(art) 
- male(bob) 
- male(chris) 
- male(daniel) 
- female(bea) 
- female(coe) 
- female(daisy) 

View: 
- person(art) 
- person(bob) 
- person(chris) 
- person(daniel) 
- person(bea) 
- person(coe) 
- person(daisy)

Note that the same relation can appear in the head of more than one rule. For example, the person relation is true of a person \( Y \) if there is an \( X \) such that \( X \) is the parent of \( Y \) *or* if \( Y \) is the parent of some person \( Z \). Note that in this case the conditions are disjunctive (at least one must be true), whereas the conditions in the grandfather case are conjunctive (both must be true).
### Ancestors:

- `ancestor(X,Z) :- parent(X,Z)`
- `ancestor(X,Z) :- ancestor(X,Y) & ancestor(Y,Z)`

### Data:

<table>
<thead>
<tr>
<th>Parent</th>
<th>Ancestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>art</td>
<td>bob</td>
</tr>
<tr>
<td>art</td>
<td>bea</td>
</tr>
<tr>
<td>bob</td>
<td>chris</td>
</tr>
<tr>
<td>bob</td>
<td>coe</td>
</tr>
<tr>
<td>coe</td>
<td>daniel</td>
</tr>
<tr>
<td>coe</td>
<td>daisy</td>
</tr>
</tbody>
</table>

### View:

<table>
<thead>
<tr>
<th>Ancestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>art,bob</td>
</tr>
<tr>
<td>art,bea</td>
</tr>
<tr>
<td>art,chris</td>
</tr>
<tr>
<td>art,coe</td>
</tr>
<tr>
<td>art,daniel</td>
</tr>
<tr>
<td>art,daisy</td>
</tr>
<tr>
<td>bob,chris</td>
</tr>
<tr>
<td>bob,coe</td>
</tr>
<tr>
<td>bob,daniel</td>
</tr>
<tr>
<td>bob,daisy</td>
</tr>
<tr>
<td>coe,daniel</td>
</tr>
<tr>
<td>coe,daisy</td>
</tr>
</tbody>
</table>

---

A person X is an ancestor of a person Z if X is the parent of Z or if there is a person Y such that X is an ancestor of and Y is an ancestor of Z. This example shows that it is possible for a relation to appear in its own definition. (But recall our discussion of stratification for a restriction on this capability.)
A childless person is one who has no children. We can define the property of being childless with the rules shown below. The first rule states that a person X is childless if X is a person and it is not the case that X is a parent. The second rule says that isparent is true of X if X is the parent of some person Y.

Note the use of the helper relation isparent here. It is tempting to write the childless rule as childless(X) :- person(X) & ~parent(X,Y). However, this would be wrong. This would define X to be childless if X is a person and there is some Y such that X is ~parent(X,Y) is true. But we really want to say that ~parent(X,Y) holds for all Y. Defining isparent and using its negation in the definition of childless allows us to express this "universal quantification".
In our development thus far, we have assumed that the extension of an n-ary relation may be any set of n-tuples from the domain. This is rarely the case. Often, there are *constraints* that limit the set of possibilities. For example, a person cannot be his own parent. In some cases, constraints involve multiple relations. For example, all parents are adults; in other words, if an entity appears in the first column of the parent relation, it must also appear as an entry in the adult relation.

In many database texts, constraints are written in direct form – by writing rules that say, in effect, that if certain things are true in an extension, then other things must also be true. The *inclusion dependency* mentioned above is an example – if an entity appears in the first column of the parent relation, it must also appear as an entry in the adult relation.

In what follows, we use a slightly less direct approach – we encode limitations by writing rules that say when a database is *not* well-formed. We simply invent a new 0-ary relation, here called illegal, and define it to be true in any extension that does not satisfy our constraints.
Example

Non-reflexivity of parenthood

illegal :- parent(X,X)

Data:
parent(art,bob)
parent(art,bea)
parent(bea,chris)
parent(bea,coe)
parent(coe,daniel)
parent(coe,daisy)

This approach works particularly well for consistency constraints like the one stating that a person cannot be his own parent.
Mutual Exclusion of gender

illegal :- male(X) & female(X)

Data:

male(art)
male(bob)
male(chris)
male(daniel)
female(bea)
female(coe)
female(daisy)

It also works well for *mutual exclusion* constraints like the one below, which states that a person cannot be in both the male and the female relations.
Inclusion Dependency on parenthood and adulthood

\[
\text{illegal} \leftarrow \text{parent}(X,Y) \& \neg \text{adult}(X)
\]

Data:

<table>
<thead>
<tr>
<th>parent(art,bob)</th>
<th>adult(art)</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent(art,bea)</td>
<td>adult(bob)</td>
</tr>
<tr>
<td>parent(bea,chris)</td>
<td>adult(bea)</td>
</tr>
<tr>
<td>parent(bea,coe)</td>
<td>adult(chris)</td>
</tr>
<tr>
<td>parent(coe,daniel)</td>
<td>adult(coe)</td>
</tr>
<tr>
<td>parent(coe,daisy)</td>
<td></td>
</tr>
</tbody>
</table>

Using this technique, we can also write the *inclusion dependency* mentioned earlier. There is an error if an entity is in the first column of the parent relation and it does not occur in the adult relation.
In updating a database, a user specifies a sentence to add to a database or a sentences to delete. In some cases, the user can group several changes of this sort in a single, so-called, atomic transaction. If the result of executing the transaction satisfies the constraints, the update is performed; otherwise it is rejected.

Unfortunately, if a user forgets to include an addition or deletion required by the constraints, this can lead to errors. In order to simplify the update process for the user, some database systems provide the administrator the ability to write update rules, i.e. rules that are executed by the system to augment a specified transaction with the additions and deletions necessary to avoid errors. In what follows, we show one way that this can be done.
Our update language includes four special operators – pluss, minus, pos, and neg. pluss takes a sentence as argument and is true if and only if the "user" specifies that sentence as an addition in a transaction. minus takes a sentence as argument and is true if and only if the "user" specifies that sentence as an addition in a transaction. pos takes a sentence as argument and is true if and only if the "system" concludes that the specified sentence should be added to the database. neg takes a sentence as argument and is true if and only if the "system" concludes that the specified sentence should be added to the database. Update rules are rules that define pos and neg in terms of pluss and minus and the current state of the database.

As an example of this mechanism in action, consider the rules shown below. The first dictates that the system remove a sentence of the form male(X) whenever the user adds a sentence of the form female(X). The second rule is analogous to the first with male and female reversed. Together, these two rules enforce the mutual exclusion on male and female.
Fathers and Mothers:

\[ \text{pos(adult}(X) \rangle \leftarrow \text{pluss} (\text{parent}(X,Y)) \]

Data:                      Update:
parent(art,bob)            pluss(parent(chris,don)
parent(art,bea)
parent(bea,chris)           pos(adult(chris))
parent(bea,coe)
parent(coe,daniel)
parent(coe,daisy)

Example

Similarly, we can enforce the inclusion dependency on parent and adult by writing the rule here. The upshot is that, if the user adds a sentence like parent(chris,don), then the system also adds a sentence of the form adult(chris).
**Fathers and Mothers:**

\[
\begin{align*}
\text{father}(X,Y) &::= \text{parent}(X,Y) \land \text{male}(X) \\
\text{mother}(X,Y) &::= \text{parent}(X,Y) \land \text{female}(X)
\end{align*}
\]

**Update Rules:**

\[
\begin{align*}
\text{pos}(\text{father}(X,Y)) &::= \\
&\text{pluss}(\text{parent}(X,Y)) \land \text{male}(X) \land \neg \text{minus}(\text{male}(X)) \\
\text{pos}(\text{father}(X,Y)) &::= \\
&\text{parent}(X,Y) \land \text{pluss}(\text{male}(X)) \land \neg \text{minus}(\text{parent}(X,Y)) \\
\text{pos}(\text{father}(X,Y)) &::= \\
&\text{pluss}(\text{parent}(X,Y)) \land \text{pluss}(\text{male}(X)) \\
\text{neg}(\text{father}(X,Y)) &::= \text{minus}(\text{parent}(X,Y)) \\
\text{neg}(\text{father}(X,Y)) &::= \text{minus}(\text{male}(X))
\end{align*}
\]

Another use of this update mechanism is to maintain materialized views. (A materialized view is a defined relation that is stored explicitly in the database, usually to save recomputation.)

Suppose, for example, we were to materialize the father relation defined earlier. Then we could write the update rules to maintain this materialized view. According to the first rule, the system should add a sentence of the form \(\text{father}(X,Y)\) whenever the user adds \(\text{parent}(X,Y)\) and \(\text{male}(X)\) is know to be true and the user does not delete that fact. The other rules cover the other cases.
Issues
A rule is *safe* if and only if every variable in the head appears in some positive subgoal in the body.

**Safe Rule:**
\[ r(X, Z) \leftarrow p(X, Y) \land q(Y, Z) \land \neg r(X, Y) \]

**Unsafe Rule:**
\[ r(X, Z) \leftarrow p(X, Y) \land q(Y, X) \]

**Unsafe Rule:**
\[ r(X, Y) \leftarrow p(X, Y) \land \neg q(Y, Z) \]

We require all rules to be safe.

The first of these restrictions is called safety. A rule in a logic program is safe if and only if every variable that appears in the head or in any negative literal in the body also appears in at least one positive literal in the body. The first rule shown here is safe. Variables X and Z appear in the head and Y appears in a negative subgoal. Fortunately, all three of those variables appear in positive subgoals as well, and so the rule is safe. The second is not safe because variable Z appears in the head but not in any positive subgoal. The third rule is not safe because the variable Z appears in a negative subgoal but not in a positive subgoal. In deductive databases, we require all rules to be safe.
The dependency graph for a set of rules is a directed graph in which (1) the nodes are the relations mentioned in the head and bodies of the rules and (2) there is an arc from a node $p$ to a node $q$ whenever $p$ occurs with the body of a rule in which $q$ is in the head.

\[
\begin{align*}
  r(X, Y) & :\leftarrow p(X, Y) \land q(X, Y) \\
  s(X, Y) & :\leftarrow r(X, Y) \\
  s(X, Z) & :\leftarrow r(X, Y) \land t(Y, Z) \\
  t(X, Z) & :\leftarrow s(X, Y) \land s(Y, X)
\end{align*}
\]

A set of rules is recursive if it contains a cycle. Otherwise, it is non-recursive.

The next two restrictions on GDL descriptions have to do with recursion. The restrictions are best defined using the notion of dependency graphs. The dependency graph for a set of rules is a directed graph in which (1) the nodes are the relations mentioned in the head and bodies of the rules and (2) there is an arc from a node $p$ to a node $q$ whenever $p$ occurs with the body of a rule in which $q$ is in the head. A set of rules is recursive if and only if its dependency graph contains a cycle.
A negation in a set of rules is said to be *stratified* if and only if there is no recursive cycle in the dependency graph involving a negation. For example, the first rule set shown here is not stratified because there is a cycle involving a negative occurrence of \( r \). By contrast, the second set of rules is stratified. The rule set is recursive, but there is no negation in the cycle. The only negative occurrence of \( r \) occurs in the definition of \( t \) and is not part of any recursion. In deductive databases, we require all negations to be stratified.
In many practical logic programming languages, mathematical functions are represented as relations. For example, the binary addition operator + is often represented by the ternary relation constant plus. For example, the rule shown here defines the combined age of two people.
Aggregates:
  countofall
  sumofall
  avgofall

Example:

    grandchildren(X,N) :-
        person(X) &
        countofall(Z,grandparent(X,Z),N)

NB: Only variables allowed in condition are aggregation variable or variables bound in other positive subgoals.

There are also aggregate operators such as countofall, sumofall, and avgofall. For example the rule here defines the number of a person's grandchildren using the countofall operator. N is the number of grandchildren of X if N is the count of all Z such that X is the grandparent of Z.