Chapter 6

Logic and Databases

This chapter discusses the relationship between logic programs and relational databases. It is demonstrated how logic can be used to represent — on a conceptual level — not only *explicit* data, but also *implicit* data (corresponding to *views* in relational database theory) and how it can be used as a *query language* for retrieval of information in a database. We do not concern ourselves with implementation issues but only remark that SLD-resolution does not necessarily provide the best inference mechanism for full logical databases. (An alternative approach is discussed in Chapter 15.) On the other hand, logic not only provides a uniform language for representation of databases — its additional expressive power also enables description, in a concise and intuitive way, of more complicated relations — for instance, relations which exhibit certain common properties (like transitivity) and relations involving structured data objects.

6.1 Relational Databases

As indicated by the name, the mathematical notion of relation is a fundamental concept in the field of relational databases. Let $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$ be collections of symbols called *domains*. In the context of database theory the domains are usually assumed to be finite although, for practical reasons, they normally include an infinite domain of numerals. In addition, the members of the domains are normally assumed to be atomic or indivisible — that is, it is not possible to access a proper part of a member.

A *database relation* R over the domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ is a subset of $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n$. R is in this case said to be n-ary. A *relational database* is a finite number of such (finite) relations. Database relations and domains will be denoted by identifiers in capital letters.

Example 6.1 Let $MALE := \{adam, bill\}$, $FEMALE := \{anne, beth\}$ and finally $PERSON := MALE \cup FEMALE$. Then:

$$MALE \times PERSON = \left\{ \begin{array}{ll} \langle adam, adam\rangle & \langle bill, adam\rangle \\ \langle adam, bill\rangle & \langle bill, bill\rangle \\ \langle adam, anne\rangle & \langle bill, anne\rangle \\ \langle adam, beth\rangle & \langle bill, beth\rangle \end{array} \right\}$$

Now, let *FATHER*, *MOTHER* and *PARENT* be relations over the domains $MALE \times PERSON$, $FEMALE \times PERSON$ and $PERSON \times PERSON$ defined as follows:

$$\begin{array}{lcl} FATHER & := & \{\langle adam, bill\rangle, \langle adam, beth\rangle\} \\ MOTHER & := & \{\langle anne, bill\rangle, \langle anne, beth\rangle\} \\ PARENT & := & \{\langle adam, bill\rangle, \langle adam, beth\rangle, \langle anne, bill\rangle, \langle anne, beth\rangle\} \end{array}$$

∎

It is of course possible to imagine alternative syntactic representations of these relations. For instance in the form of tables:

FATHER:

C_1	C_2
adam	*bill*
adam	*beth*

MOTHER:

C_1	C_2
anne	*bill*
anne	*beth*

PARENT:

C_1	C_2
adam	*bill*
adam	*beth*
anne	*bill*
anne	*beth*

or as a collection of labelled tuples (that is, facts):

$$\begin{array}{l} father(adam, bill). \\ father(adam, beth). \\ mother(anne, bill). \\ mother(anne, beth). \\ parent(adam, bill). \\ parent(adam, beth). \\ parent(anne, bill). \\ parent(anne, beth). \end{array}$$

The table-like representation is the one found in most textbooks on relational databases whereas the latter is a logic program. The two representations are isomorphic if no notice is taken of the names of the columns in the tables. Such names are called *attributes* and are needed only to simplify the specification of some of the operations discussed in Section 6.3. It is assumed that the attributes of a table are distinct. In what follows the notation $R(A_1, A_2, \ldots, A_n)$ will be used to describe the name, R, and attributes, $\langle A_1, A_2, \ldots, A_n\rangle$, of a database table (i.e. relation). $R(A_1, A_2, \ldots, A_n)$ is sometimes called a *relation scheme*. When not needed, the attributes are omitted and a table will be named only by its relation-name.

A major difference between the two representations which is not evident above, is the set of values which may occur in each column/argument-position of the representations. Logic programs have only a single domain consisting of terms and the user is permitted to write:

$$father(anne, adam).$$

whereas in a relational database this is usually not possible since $anne \notin MALE$. To avoid such problems a notion of *type* is needed.

Despite this difference it should be clear that any relational database can be represented as a logic program (where each domain of the database is extended to the set of all terms) consisting solely of ground facts. Such a set of facts is commonly called the *extensional database* (EDB).

6.2 Deductive Databases

After having established the relationship between relational databases and a (very simple) class of logic programs, different extensions to the relational database-model are studied. We first consider the use of variables and a simple form of rules. By such extensions it is possible to describe — in a more succinct and intuitive manner — many database relations. For instance, using rules and variables the database above can be represented by the program:

$$
\begin{aligned}
&parent(X, Y) \leftarrow father(X, Y).\\
&parent(X, Y) \leftarrow mother(X, Y).\\
&father(adam, bill).\\
&father(adam, beth).\\
&mother(anne, bill).\\
&mother(anne, beth).
\end{aligned}
$$

The part of a logic program which consists of rules and nonground facts is called the *intensional database* (IDB). Since logic programs facilitate definition of new atomic formulas which are ultimately *deduced* from explicit facts, logic programs are often referred to as *deductive databases*. The logic programs above are also examples of a class of logic programs called *datalog* programs. They are characterized by the absence of functors. In other words, the set of terms used in the program solely consists of constant symbols and variables. For the representation of relational databases this is sufficient since the domains of the relations are assumed to be finite and it is therefore always possible to represent the individuals with a finite set of constant terms. In the last section of this chapter logic programs which make also use of compound terms are considered, but until then our attention will be restricted to datalog programs.

Example 6.2 Below is given a deductive family-database whose extensional part consists of definitions of $male/1$, $female/1$, $father/2$ and $mother/2$ and whose intensional part consists of $parent/2$ and $grandparent/2$:

$$grandparent(X, Z) \leftarrow parent(X, Y), parent(Y, Z).$$

$$
\begin{aligned}
&parent(X, Y) \leftarrow father(X, Y).\\
&parent(X, Y) \leftarrow mother(X, Y).
\end{aligned}
$$

$$
\begin{array}{ll}
father(adam, bill). & mother(anne, bill).\\
father(adam, beth). & mother(anne, beth).\\
father(bill, cathy). & mother(cathy, donald).\\
father(donald, eric). & mother(diana, eric).
\end{array}
$$

$$\begin{array}{ll} \mathit{female}(\mathit{anne}). & \mathit{male}(\mathit{adam}). \\ \mathit{female}(\mathit{beth}). & \mathit{male}(\mathit{bill}). \\ \mathit{female}(\mathit{cathy}). & \mathit{male}(\mathit{donald}). \\ \mathit{female}(\mathit{diana}). & \mathit{male}(\mathit{eric}). \end{array}$$

∎

In most cases it is possible to organize the database in many alternative ways. Which organization to choose is of course highly dependent on what information one needs to retrieve. Moreover, it often determines the size of the database. Finally, in the case of updates to the database, the organization is very important to avoid inconsistencies in the database — for instance, how should the removal of the labelled tuple $\mathit{parent}(\mathit{adam}, \mathit{bill})$ from the database in Example 6.2 be handled? Although updates are essential in a database system they will not be discussed in this book.

Another thing worth noticing about Example 6.2 is that the unary definitions $\mathit{male}/1$ and $\mathit{female}/1$ can be seen as *type declarations*. It is easy to add another such type declaration for the domain of persons:

$$\begin{array}{l} \mathit{person}(X) \leftarrow \mathit{male}(X). \\ \mathit{person}(X) \leftarrow \mathit{female}(X). \end{array}$$

It is now possible to "type" e.g. the database on page 103 by adding to the body of every clause the type of each argument in the head of the clause:

$$\begin{array}{l} \mathit{parent}(X, Y) \leftarrow \mathit{person}(X), \mathit{person}(Y), \mathit{father}(X, Y). \\ \mathit{parent}(X, Y) \leftarrow \mathit{person}(X), \mathit{person}(Y), \mathit{mother}(X, Y). \\ \\ \mathit{father}(\mathit{adam}, \mathit{bill}) \leftarrow \mathit{male}(\mathit{adam}), \mathit{person}(\mathit{bill}). \\ \mathit{father}(\mathit{adam}, \mathit{beth}) \leftarrow \mathit{male}(\mathit{adam}), \mathit{person}(\mathit{beth}). \\ \quad \vdots \\ \\ \mathit{person}(X) \leftarrow \mathit{male}(X). \\ \mathit{person}(X) \leftarrow \mathit{female}(X). \\ \quad \vdots \end{array}$$

In this manner, "type-errors" like $\mathit{father}(\mathit{anne}, \mathit{adam})$ may be avoided.

6.3 Relational Algebra vs. Logic Programs

In database textbooks one often encounters the concept of *views*. A view can be thought of as a relation which is not explicitly stored in the database, but which is created by means of operations on existing database relations and other views. Such implicit relations are described by means of some *query-language* which is often compiled into *relational algebra* for the purpose of computing the views. Below it will be shown that all standard operations of relational algebra can be mimicked in logic programming (with negation) in a natural way. The objective of this section is twofold — first it shows that logic programs have at least the computational power of relational algebra. Second, it also provides an alternative to SLD-resolution as the operational semantics of a class of logic programs.

The primitive operations of relational algebra are *union*, *set difference*, *cartesian product*, *projection* and *selection*.

Given two n-ary relations over the same domains, the *union* of the two relations, R_1 and R_2 (denoted $R_1 \cup R_2$), is the set:

$$\{\langle x_1, \ldots, x_n\rangle \mid \langle x_1, \ldots, x_n\rangle \in R_1 \vee \langle x_1, \ldots, x_n\rangle \in R_2\}$$

Using definite programs the union of two relations — represented by the predicate symbols r_1/n and r_2/n — can be specified by the two rules:

$$r(X_1, \ldots, X_n) \leftarrow r_1(X_1, \ldots, X_n).$$
$$r(X_1, \ldots, X_n) \leftarrow r_2(X_1, \ldots, X_n).$$

For instance, if the EDB includes the definitions *father*/2 and *mother*/2, then *parent*/2 can be defined as the union of the relations *father*/2 and *mother*/2:[1]

$$parent(X, Y) \leftarrow father(X, Y).$$
$$parent(X, Y) \leftarrow mother(X, Y).$$

The *difference* $R_1 \setminus R_2$ of two relations R_1 and R_2 over the same domains yields the new relation:

$$\{\langle x_1, \ldots, x_n\rangle \in R_1 \mid \langle x_1, \ldots, x_n\rangle \notin R_2\}$$

In logic programming it is not possible to define such relations without the use of negation; however, using negation it may be defined thus:

$$r(X_1, \ldots, X_n) \leftarrow r_1(X_1, \ldots, X_n), not\ r_2(X_1, \ldots, X_n).$$

For example, let *parent*/2 and *mother*/2 belong to the EDB. Now, *father*/2 can be defined as the difference of the relations *parent*/2 and *mother*/2:

$$father(X, Y) \leftarrow parent(X, Y), not\ mother(X, Y).$$

The *cartesian product* of two relations R_1 and R_2 (denoted $R_1 \times R_2$) yields the new relation:

$$\{\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle \mid \langle x_1, \ldots, x_m\rangle \in R_1 \wedge \langle y_1, \ldots, y_n\rangle \in R_2\}$$

Notice that R_1 and R_2 may have both different domains and different arities. Moreover, if R_1 and R_2 contain disjoint sets of attributes they are carried over to the resulting relation. However, if the original relations contain some joint attribute the attribute of the two columns in the new relation must be renamed into distinct ones. This can be done e.g. by prefixing the joint attributes in the new relation by the relation where they came from. For instance, in the relation $R(A, B) \times S(B, C)$ the attributes are, from left to right, A, $R.B$, $S.B$ and C. Obviously, it is possible to achieve the same effect in other ways.

In logic programming the cartesian product is mimicked by the rule:

$$r(X_1, \ldots, X_m, Y_1, \ldots, Y_n) \leftarrow r_1(X_1, \ldots, X_m), r_2(Y_1, \ldots, Y_n).$$

[1] In what follows we will sometimes, by abuse of language, write "the relation p/n". Needless to say, p/n is not a relation but a predicate symbol which *denotes* a relation.

For instance, let $male/1$ and $female/1$ belong to the EDB. Then the set of all male-female couples can be defined by the rule:

$$couple(X, Y) \leftarrow male(X), female(Y).$$

Projection can be seen as the deletion and/or rearrangement of one or more "columns" of a relation. For instance, by projecting the F- and C-attributes of the relation $FATHER(F, C)$ on the F-attribute (denoted $\pi_F(FATHER(F, C))$) the new relation:

$$\{\langle x_1 \rangle \mid \langle x_1, x_2 \rangle \in FATHER\}$$

is obtained. The same can be achieved in Prolog by means of the rule:

$$father(X) \leftarrow father(X, Y).$$

The *selection* of a relation R is denoted $\sigma_F(R)$ (where F is a formula) and is the set of all tuples $\langle x_1, \ldots, x_n \rangle \in R$ such that "F is true for $\langle x_1, \ldots, x_n \rangle$". How to translate such an operation to a logic program depends on the appearance of the constraining formula F. In general F is only allowed to contain atomic objects, attributes, $\wedge$, $\vee$, $\neg$ and some simple comparisons (e.g. "=" and "<"). For instance, the database relation defined by $\sigma_{Y \geq 1,000,000}\ INCOME(X, Y)$ may be defined as follows in Prolog:

$$millionaire(X, Y) \leftarrow income(X, Y), Y \geq 1000000.$$

Some other operations (like intersection and composition) are sometimes encountered in relational algebra but they are usually all defined in terms of the mentioned, primitive ones and are therefore not discussed here. However, one of them deserves special attention — namely the *natural join*.

The natural join of two relations R and S can be computed only when the columns are named by attributes. Thus, assume that $T_1, \ldots, T_k$ are the attributes which appear both in R and in S. Then the natural join of R and S is defined thus:

$$R \bowtie S := \pi_A\ \sigma_{R.T_1 = S.T_1\ \wedge \cdots \wedge\ R.T_k = S.T_k}\ (R \times S)$$

where A is the list of all attributes of $R \times S$ with exception of $S.T_1, \ldots, S.T_k$. Thus, the natural join is obtained by (1) taking the cartesian product of the two relations, (2) selecting those tuples which have identical values in the columns with the same attribute and (3) filtering out the superfluous columns. Notice that if R and S have disjoint sets of attributes, then the natural join reduces to an ordinary cartesian product.

To illustrate the operation, consider the relation defined by $F(X, Y) \bowtie P(Y, Z)$ where $F(X, Y)$ and $P(Y, Z)$ are defined according to Figure 6.1(a) and 6.1(b) and denote the relation between fathers/parents and their children.

Now $F(X, Y) \bowtie P(Y, Z)$ is defined as $\pi_{X, F.Y, Z}\ \sigma_{F.Y = P.Y}\ (F(X, Y) \times P(Y, Z))$. Hence the first step consists in computing the cartesian product $F(X, Y) \times P(Y, Z)$ (cf. Figure 6.1(c)). Next the tuples with equal values in the columns named by $F.Y$ and $P.Y$ are selected (Figure 6.1(d)). Finally this is projected on the X, $F.Y$ and Z attributes yielding the relation in Figure 6.1(e).

If we assume that $father/2$ and $parent/2$ are used to represent the database relations F and P then the same relation may be defined with a single definite clause as follows:

X	Y
adam	*bill*
bill	*cathy*

(a)

Y	Z
adam	*bill*
bill	*cathy*
cathy	*dave*

(b)

X	$F.Y$	$P.Y$	Z
adam	*bill*	*adam*	*bill*
adam	*bill*	*bill*	*cathy*
adam	*bill*	*cathy*	*dave*
bill	*cathy*	*adam*	*bill*
bill	*cathy*	*bill*	*cathy*
bill	*cathy*	*cathy*	*dave*

(c)

X	$F.Y$	$P.Y$	Z
adam	*bill*	*bill*	*cathy*
bill	*cathy*	*cathy*	*dave*

(d)

X	$F.Y$	Z
adam	*bill*	*cathy*
bill	*cathy*	*dave*

(e)

Figure 6.1: Natural join

$$grandfather(X, Y, Z) \leftarrow father(X, Y), parent(Y, Z).$$

Notice that the standard definition of *grandfather*/2:

$$grandfather(X, Z) \leftarrow father(X, Y), parent(Y, Z).$$

is obtained by projecting $X, F.Y, Z$ on X and Z, that is, by performing the operation $\pi_{X,Z}(F(X, Y) \bowtie P(Y, Z))$.

6.4 Logic as a Query-language

In the previous sections it was observed that logic provides a uniform language for representing both explicit data and implicit data (so-called views). However, deductive databases are of little or no interest if it is not possible to retrieve information from the database. In traditional databases this is achieved by so-called *query-languages*. Examples of existing query-languages for relational databases are e.g. ISBL, SQL, QUEL and Query-by-Example.

By now it should come as no surprise to the reader that logic programming can be used as a query-language in the same way it was used to define views. For instance, to retrieve the children of Adam from the database in Example 6.2 one only has to give the goal clause:

$$\leftarrow parent(adam, X).$$

To this Prolog-systems would respond with the answers $X = bill$ and $X = beth$, or put alternatively — the unary relation $\{\langle bill\rangle, \langle beth\rangle\}$. Likewise, in response to the goal:

$$\leftarrow mother(X, Y).$$

Prolog produces four answers:

$$\begin{array}{ll} X = anne, & Y = bill \\ X = anne, & Y = beth \\ X = cathy, & Y = donald \\ X = diana, & Y = eric \end{array}$$

That is, the relation:

$$\{\langle anne, bill\rangle, \langle anne, beth\rangle, \langle cathy, donald\rangle, \langle diana, eric\rangle\}$$

Notice that a failing goal (e.g. $\leftarrow parent(X, adam)$) computes the empty relation as opposed to a succeeding goal without variables (e.g $\leftarrow parent(adam, bill)$) which computes a singleton relation containing a 0-ary tuple.

Now consider the following excerpt from a database:

$$\begin{array}{l} likes(X, Y) \leftarrow baby(Y). \\ baby(mary). \\ \quad\vdots \end{array}$$

Informally the two clauses say that "Everybody likes babies" and "Mary is a baby". Consider the result of the query "Is anyone liked by someone?". In other words the goal clause:

$$\leftarrow likes(X, Y).$$

Clearly Prolog will reply with $Y = mary$ and X being unbound. This is interpreted as "Everybody likes Mary" but what does it mean in terms of a database relation? One solution to the problem is to declare a type-predicate and to extend the goal with calls to this new predicate:

$$\leftarrow likes(X, Y), person(X), person(Y).$$

In response to this goal Prolog would enumerate all individuals of type $person/1$. It is also possible to add the extra literal $person(X)$ to the database rule. Another approach which is often employed when describing deductive databases is to adopt certain assumptions about the world which is modelled. One such assumption was mentioned already in connection with Chapter 4 — namely the closed world assumption (CWA). Another assumption which is usually adopted in deductive databases is the so-called *domain closure assumption* (DCA) which states that "the only existing individuals are those mentioned in the database". In terms of logic this can be expressed through the additional axiom:

$$\forall X(X = c_1 \vee X = c_2 \vee \cdots \vee X = c_n)$$

where $c_1, c_2, \ldots, c_n$ are all the constants occurring in the database. With this axiom the relation defined by the goal above becomes $\{\langle t, mary\rangle \mid t \in U_P\}$. However, this assumes that the database contains no functors and only a finite number of constants.

6.5 Special Relations

The main objective of this section is to show how to define relations that possess certain properties occurring frequently both in real life and in mathematics. This includes properties like *reflexivity*, *symmetry* and *transitivity*.

Let R be a binary relation over some domain $\mathcal{D}$. Then:

- R is said to be *reflexive* iff for all $x \in \mathcal{D}$, it holds that $\langle x, x \rangle \in R$;
- R is *symmetric* iff $\langle x, y \rangle \in R$ implies that $\langle y, x \rangle \in R$;
- R is *anti-symmetric* iff $\langle x, y \rangle \in R$ and $\langle y, x \rangle \in R$ implies that $x = y$;
- R is *transitive* iff $\langle x, y \rangle \in R$ and $\langle y, z \rangle \in R$ implies that $\langle x, z \rangle \in R$;
- R is *asymmetric* iff $\langle x, y \rangle \in R$ implies that $\langle y, x \rangle \notin R$.

To define an EDB which possesses one of these properties is usually a rather cumbersome task if the domain is large. For instance, to define a reflexive relation over a domain with n elements requires n tuples, or n facts in the case of a logic program. Fortunately, in logic programming, relations can be defined to be reflexive with a single clause of the form:

$$r(X, X).$$

However, in many cases one thinks of the Herbrand universe as the coded union of several domains. For instance, the Herbrand universe consisting of the constants *bill*, *kate* and *love* may be thought of as the coded union of persons and abstract notions. If — as in this example — the intended domain of $r/2$ (encoded as terms) ranges over proper subsets of the Herbrand universe and if the type predicate $t/1$ characterize this subset, a reflexive relation can be written as follows:

$$r(X, X) \leftarrow t(X).$$

For instance, in order to say that "every person looks like himself" we may write the following program:

$$\begin{array}{l} looks_like(X, X) \leftarrow person(X). \\ person(bill). \\ person(kate). \\ abstract(love). \end{array}$$

In order to define a symmetric relation R it suffices to specify only one of the pairs $\langle x, y \rangle$ and $\langle y, x \rangle$ if $\langle x, y \rangle \in R$. Then the program is extended with the rule:

$$r(X, Y) \leftarrow r(Y, X).$$

However, as shown below such programs suffer from operational problems.

Example 6.3 Consider the domain:

$$\{sarah, diane, pamela, simon, david, peter\}$$

The relation "... is married to ..." clearly is symmetric and it may be written either as an extensional database:

$$\begin{array}{l} married(sarah, simon). \\ married(diane, david). \\ married(pamela, peter). \\ married(simon, sarah). \\ married(david, diane). \\ married(peter, pamela). \end{array}$$

or more briefly as a deductive database:

$$\begin{array}{l} married(X, Y) \leftarrow married(Y, X). \\ married(sarah, simon). \\ married(diane, david). \\ married(pamela, peter). \end{array}$$

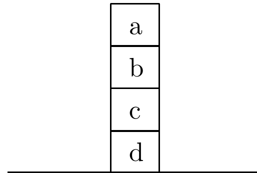
∎

Transitive relations can also be simplified by means of rules. Instead of a program P consisting solely of facts, P can be fully described by the clause:

$$r(X, Z) \leftarrow r(X, Y), r(Y, Z).$$

together with all $r(a, c) \in P$ for which there exists no b ($b \neq a$ and $b \neq c$) such that $r(a, b) \in P$ and $r(b, c) \in P$.

Example 6.4 Consider the world consisting of the "objects" a, b, c and d:



The relation "... is positioned over ..." clearly is transitive and may be defined either through a purely extensional database:

$$\begin{array}{ll} over(a, b). & over(a, c). \\ over(a, d). & over(b, c). \\ over(b, d). & over(c, d). \end{array}$$

or alternatively as the deductive database:

$$\begin{array}{l} over(X, Z) \leftarrow over(X, Y), over(Y, Z). \\ over(a, b). \\ over(b, c). \\ over(c, d). \end{array}$$

∎

The definitions above are declaratively correct, but they suffer from operational problems when executed by Prolog systems. Consider the goal $\leftarrow married(diane, david)$ together with the deductive database of Example 6.3. Clearly $married(diane, david)$ is a logical consequence of the program but any Prolog interpreter would go into an infinite loop — first by trying to prove:

$$\leftarrow married(diane, david).$$

Via unification with the rule a new goal clause is obtained:

$$\leftarrow married(david, diane).$$

When trying to satisfy $married(david, diane)$ the subgoal is once again unified with the rule yielding a new goal, identical to the initial one. This process will obviously go on forever. The misbehaviour can, to some extent, be avoided by moving the rule textually after the facts. By doing so it may be possible to find some (or all) refutations before going into an infinite loop. However, no matter how the clauses are ordered, goals like $\leftarrow married(diane, diane)$ always lead to loops.

A better way of avoiding such problems is to use an auxiliary anti-symmetric relation instead and to take the *symmetric closure* of this relation. This can be done by renaming the predicate symbol of the EDB with the auxiliary predicate symbol and then introducing two rules which define the symmetric relation in terms of the auxiliary one.

Example 6.5 The approach is illustrated by defining $married/2$ in terms of the auxiliary definition $wife/2$ which is anti-symmetric:

$$\begin{aligned}
&married(X, Y) \leftarrow wife(X, Y).\\
&married(X, Y) \leftarrow wife(Y, X).\\
\\
&wife(sarah, simon).\\
&wife(diane, david).\\
&wife(pamela, peter).
\end{aligned}$$

This program has the nice property that it never loops — simply because it is not recursive. ∎

A similar approach can be applied when defining transitive relations. A new auxiliary predicate symbol is introduced and used to rename the EDB. Then the *transitive closure* of this relation is defined by means of the following two rules (where $p/2$ denotes the transitive relation and $q/2$ the auxiliary one):

$$\begin{aligned}
&p(X, Y) \leftarrow q(X, Y)\\
&p(X, Y) \leftarrow q(X, Z), p(Z, Y).
\end{aligned}$$

Example 6.6 The relation $over/2$ may be defined in terms of the predicate symbol $on/2$:

$$\begin{aligned}
&over(X, Y) \leftarrow on(X, Y).\\
&over(X, Z) \leftarrow on(X, Y), over(Y, Z).
\end{aligned}$$

$$on(a, b).$$
$$on(b, c).$$
$$on(c, d).$$

Notice that recursion is not completely eliminated. It may therefore happen that the program loops. As shown below this depends on properties of the auxiliary relation. ∎

The transitive closure may be combined with the *reflexive closure* of a relation. Given an auxiliary relation denoted by $q/2$, its reflexive and transitive closure is obtained through the additional clauses:

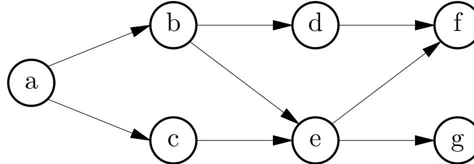
$$p(X, X).$$
$$p(X, Y) \leftarrow q(X, Y).$$
$$p(X, Z) \leftarrow q(X, Y), p(Y, Z).$$

Actually, the second clause is superfluous since it follows logically from the first and third clause: any goal, $\leftarrow p(a, b)$, which is refuted through unification with the second clause can be refuted through unification with the third clause where the recursive subgoal is unified with the first clause.

Next we consider two frequently encountered types of relations — namely *partial orders* and *equivalence relations*.

A binary relation is called a *partial order* if it is reflexive, anti-symmetric and transitive whereas a relation which is reflexive, symmetric and transitive is called an *equivalence relation*.

Example 6.7 Consider a directed, acyclic graph:



It is easy to see that the relation "there is a path from ... to ..." is a partial order given the graph above. To formally define this relation we start with an auxiliary, asymmetric relation (denoted by $edge/2$) which describes the edges of the graph:

$$edge(a, b). \qquad edge(c, e).$$
$$edge(a, c). \qquad edge(d, f).$$
$$edge(b, d). \qquad edge(e, f).$$
$$edge(b, e). \qquad edge(e, g).$$

Then the reflexive and transitive closure of this relation is described through the two clauses:

$$path(X, X).$$
$$path(X, Z) \leftarrow edge(X, Y), path(Y, Z).$$

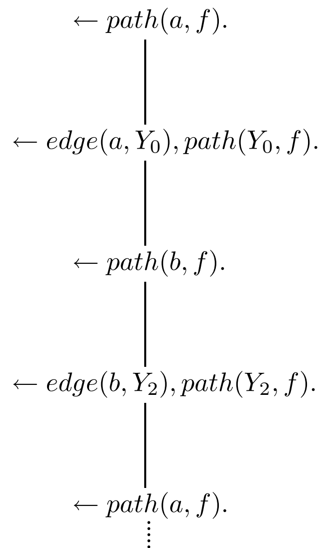


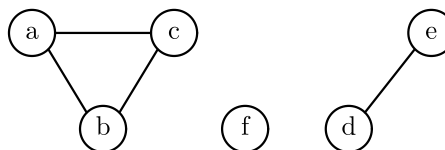
Figure 6.2: Infinite branch in the SLD-tree

This program does not suffer from infinite loops. In fact, no partial order defined in this way will loop as long as the domain is finite. However, if the graph contains a loop it may happen that the program starts looping — consider the addition of a cycle in the above graph. For instance, an additional edge from b to a:

$$edge(b, a).$$

Part of the SLD-tree of the goal $\leftarrow path(a, f)$ is depicted in Figure 6.2. The SLD-tree clearly contains an infinite branch and hence it may happen that the program starts looping without returning any answers. In Chapter 11 this problem will be discussed and a solution will be suggested. ∎

Example 6.8 Next consider some points on a map and bi-directed edges between the points:



This time the relation "there is a path from ... to ..." is an equivalence relation. To define the relation we may start by describing one half of each edge in the graph:

$$edge(a, b).$$
$$edge(a, c).$$
$$edge(b, c).$$
$$edge(d, e).$$

Next the other half of each edge is described by means of the symmetric closure of the relation denoted by $edge/2$:

$$bi_edge(X, Y) \leftarrow edge(X, Y).$$
$$bi_edge(X, Y) \leftarrow edge(Y, X).$$

Finally, $path/2$ is defined by taking the reflexive and transitive closure of this relation:

$$path(X, X).$$
$$path(X, Z) \leftarrow bi_edge(X, Y), path(Y, Z).$$

Prolog programs defining equivalence relations usually suffer from termination problems unless specific measures are taken (cf. Chapter 11). ∎

6.6 Databases with Compound Terms

In relational databases it is usually required that the domains consist of atomic objects, something which simplifies the mathematical treatment of relational databases. Naturally, when using logic programming, nothing prevents us from using structured data when writing deductive databases. This allows for data abstraction and in most cases results in greater expressive power and improves readability of the program.

Example 6.9 Consider a database which contains members of families and the addresses of the families. Imagine that a family is represented by a ternary term *family*/3 where the first argument is the name of the husband, the second the name of the wife and the last a structure which contains the names of the children. The absence of children is represented by the constant *none* whereas the presence of children is represented by the binary term of the form $c(x, y)$ whose first argument is the name of one child and whose second argument recursively contains the names of the remaining children (intuitively *none* can be thought of as the empty set and $c(x, y)$ can be thought of as a function which constructs a set by adding x to the set represented by y). An excerpt from such a database might look as follows:

$$address(family(john, mary, c(tom, c(jim, none))), main_street(3)).$$
$$address(family(bill, sue, none), main_street(4)).$$

$$parent(X, Y) \leftarrow$$
$$\quad address(family(X, Z, Children), Street),$$
$$\quad among(Y, Children).$$
$$parent(X, Y) \leftarrow$$
$$\quad address(family(Z, X, Children), Street),$$
$$\quad among(Y, Children).$$

$$husband(X) \leftarrow$$
$$\quad address(family(X, Y, Children), Street).$$

$$wife(Y) \leftarrow$$
$$\quad address(family(X, Y, Children), Street).$$

$$married(X, Y) \leftarrow$$
$$\quad address(family(X, Y, Children), Street).$$
$$married(Y, X) \leftarrow$$
$$\quad address(family(X, Y, Children), Street).$$

$$among(X, c(X, Y)).$$
$$among(X, c(Y, Z)) \leftarrow$$
$$\quad among(X, Z).$$

∎

The database above *can* be represented in the form of a traditional database by introducing a unique key for each family. For example as follows:

$$husband(f1, john).$$
$$husband(f2, bill).$$

$$wife(f1, mary).$$
$$wife(f2, sue).$$

$$child(f1, tom).$$
$$child(f1, jim).$$

$$address(f1, main_street, 3).$$
$$address(f2, main_street, 4).$$

$$parent(X, Y) \leftarrow husband(Key, X), child(Key, Y).$$
$$\vdots$$

However, the latter representation is less readable and it may also require some extra book-keeping to make sure that each family has a unique key.

To conclude — the issues discussed in this chapter were raised to demonstrate the advantages of using logic as a uniform language for representing databases. Facts, rules and queries can be written in a single language. Moreover, logic supports definition of relations via recursive rules, something which is not allowed in traditional databases. Finally, the use of structured data facilitates definition of relations which cannot be made in traditional relational databases. From this stand-point logic programming provides a very attractive conceptual framework for describing relational databases. On the other hand we have not raised important issues like how to implement such databases let alone how to handle updates to deductive databases.

Exercises

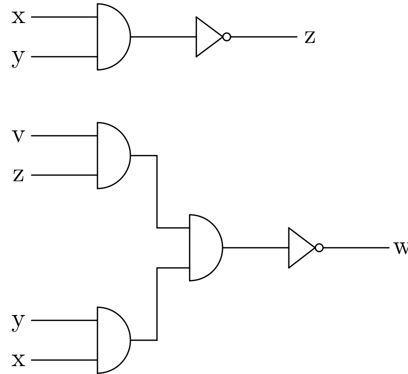
6.1 Reorganize the database in Example 6.2 so that *father*/2 and *mother*/2 become part of the intensional database.

6.2 Extend Example 6.2 with some more persons. Then define the following predicate symbols (with obvious intended interpretations):

- *grandchild*/2
- *sister*/2
- *brother*/2
- *cousins*/2
- *uncle*/2
- *aunt*/2

6.3 Consider an arbitrary planar map of countries. Write a program which colours the map using only four colours so that no two adjacent countries have the same colour. NOTE: Two countries which meet only pointwise are not considered to be adjacent.

6.4 Define the input-output behaviour of AND- and inverter-gates. Then describe the relation between input and output of the following nets:



6.5 Translate the following relational algebra expressions into definite clauses.

- $\pi_{X,Y}(HUSBAND(Key, X) \bowtie WIFE(Key, Y))$
- $\pi_X(PARENT(X, Y) \cup \pi_X\ \sigma_{Y \leq 20{,}000}\ INCOME(X, Y))$

6.6 The following clauses define a binary relation denoted by $p/2$ in terms of the relations $q/2$ and $r/2$. How would you define the same relation using relational algebra?

$$p(X, Y) \leftarrow q(Y, X).$$
$$p(X, Y) \leftarrow q(X, Z), r(Z, Y).$$

6.7 Let R_1 and R_2 be subsets of $\mathcal{D} \times \mathcal{D}$. Define the composition of R_1 and R_2 using (1) definite programs; (2) relational algebra.

6.8 Let R_1 and R_2 be subsets of $\mathcal{D} \times \mathcal{D}$. Define the intersection of R_1 and R_2 using (1) definite programs; (2) relational algebra.

6.9 An ancestor is a parent, a grandparent, a great-grandparent etc. Define a relation $ancestor/2$ which is to hold if someone is an ancestor of somebody else.

6.10 Andrew, Ann, and Adam are siblings and so are Bill, Beth and Basil. Describe the relationships between these persons using as few clauses as possible.

6.11 Define a database which relates dishes and all of their ingredients. For instance, pancakes contain milk, flour and eggs. Then define a relation which describes the available ingredients. Finally define two relations:

- $can_cook(X)$ which should hold for a dish X if all its ingredients are available;
- $needs_ingredient(X, Y)$ which holds for a dish X and an ingredient Y if X contains Y.

6.12 Modify the previous exercise as follows — add to the database the quantity available of each ingredient and for each dish the quantity needed of each ingredient. Then modify the definition of $can_cook/1$ so that the dish can be cooked if each of its ingredients is available in sufficient quantity.