Part III

Complex Queries and Reasoning

The importance of complex queries in advanced database systems cannot be overstated. At the introduction of the relational model, powerful logic-based queries were primarily motivated by their importance for end users. Subsequently, a long experience with SQL and large-scale commercial applications has shown that powerful query languages are essential in modern databases that use distributed environments, parallel machines, and client/server architectures.

Since support for complex queries means support for complex reasoning on large databases, this line of database work is also tackling problems previously addressed in research domains, such as knowledge representation, nonmonotonic reasoning, and expert systems. The next three chapters provide a unified introduction to the complex field of database and knowledge-based systems. In Chapter 8, we revisit relational query languages and extend them with more powerful constructs such as recursion, complex objects, and flexible set aggregates. In Chapter 9, we discuss the implementation of these extended queries in deductive databases and SQL systems. Finally, in Chapter 10, we explore recent advances in nonmonotonic reasoning that provide a unified model for temporal reasoning, active databases, and nondeterministic queries.

Chapter 8

The Logic of Query Languages

First-order logic provides a conceptual foundation for relational query languages. This foundation was established from the very first introduction of the relational data model by E. F. Codd, who introduced the parallel notions of relational calculus and relational algebra. Relational calculus provides a logic-based model for declarative query languages; relational algebra provides its operational equivalent: safe queries in predicate calculus can be transformed into equivalent relational expressions, and vice versa. The transformation of a calculus expression into an equivalent relational algebra expression represents the first step in efficient query implementation and optimization.

However, relational calculus has limited expressive power and cannot express many important queries, such as transitive closures and generalized aggregates. This situation has led to the design of more powerful logic-based languages that subsume relational calculus. First among these is the rule-based language Datalog, which is the focus of a large body of research and also of this chapter.

8.1 Datalog

In a Datalog representation, the database is viewed as a set of facts, one fact for each tuple in the corresponding table of the relational database, where the name of the relation becomes the predicate name of the fact. For instance, the facts in Example 8.2 correspond to the relational database of Example 8.1.

Example 8.1 A relational database about students and the courses they took

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

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

Example 8.2 The Datalog equivalent of Example 8.1

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

took('Joe Doe', cs123, 2.7).
took('Jim Jones', cs101, 3.0).
took('Jim Jones', cs143, 3.3).
took('JimBlack', cs143, 3.3).
took('JimBlack', cs101, 2.7).
```

A *fact* is a logical predicate having only constants (i.e., no variables) as its arguments. We will use the accepted convention of denoting constants by tokens that begin with lowercase characters or numbers, while denoting variables by tokens that begin with uppercase. Thus, in a predicate such as

```
took(Name, cs143, Grade)
```

`Name` and `Grade` denote variables, while `cs143` denotes a constant. However, tokens in quotes, such as `/qt7Jim Black/qt7`, denote constants. Also, `Name`, `cs143`, and `Grade` are, respectively, the first, second, and third argument of the ternary predicate `took`. Both `student` and `took` are three-argument predicates, or equivalently, ternary predicates, or predicates with arity 3.

Rules constitute the main construct of Datalog programs. For instance, Example 8.3 defines all students at the junior level who have taken cs101 and cs143. Thus, `firstreq(Name)` is the head; `student(Name, Major, junior)`, `took(Name, cs101, Grade1)`, and `took(Name, cs143, Grade2)` are, respectively, the first, second, and third goal of the rule. Together, these three goals form the body of the rule.

Example 8.3 Find the name of junior-level students who have taken both cs101 and cs143

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

The commas separating the goals stand for logical conjuncts. Therefore, the order in which the goals appear in the rule is immaterial. Since the commas separating the goals stand for logical AND, the symbols "∧" and "&" are often used in their place. Another common notational variation is the use of the symbol ":-" instead of the arrow to separate the head from the body of the rule.

A logical disjunct is represented via multiple rules with the same head predicate (i.e., sharing the same predicate name and arity). Thus, to find those juniors who took either course cs131 or course cs151, with grade better than 3.0, we would write the following:

Example 8.4 Junior-level students who took course cs131 or course cs151 with grade better than 3.0

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

Observe that in the first rule of Example 8.4, the variable `Major` occurs only once; therefore, it can be replaced with the symbol "_", which is called an *anonymous* variable, and stands for a uniquely named variable that does not appear anywhere else in the rule (see the second rule of Example 8.4). The set of rules having as their heads a predicate with the same name p is called the definition of p. Thus, the definition of a derived predicate is similar to the definition of a virtual view in relational databases. The meaning of such a definition is independent of the order in which these rules are listed, and independent of the order in which the goals appear in the rules. Table 8.1 displays the corresponding nomenclatures of Datalog and the relational model. Therefore, *base predicates* correspond to database relations and are defined by the database schema, while *derived predicates* are defined by rules. It is also common to use the terms *extensional database* and *intensional database* to refer to base predicates and derived predicates, respectively. In deductive databases, the assumption normally made is that these two form disjoint sets: that is, base predicates never appear in the heads of rules.

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

Table 8.1: The terminology of Datalog versus the relational model

Since rules are merely definitional devices, concrete Datalog programs also contain one or more query goals to specify which of the derived relations must actually be computed. Query goals can have different forms. A query that contains no variables is called a *boolean query* or a *closed query*; the answer to such a query is either yes or no. For instance,

?`firstreq('Jim Black')`

is a closed query with answer yes or no depending on whether '`Jim Black`' has satisfied the first requirement. On the other hand, consider the goal

?`firstreq(X)`

Since `X` is a variable, the answer to this query is a (possibly empty) set of facts for the students who satisfy the first requirement, as follows:

`firstreq('Jim Jones')`
`firstreq('Jim Black')`

In general, query goals will mix variables and constants in their arguments.

Rules represent a powerful formalism from both theoretical and practical viewpoints. Their practical appeal follows from the ability to view goals in rules as search patterns. For instance, in the second rule of Example 8.4, we are searching for `took` tuples with `cs151` as their second argument, and a grade greater than 3.0. Also, we are looking for the pattern `junior` in the third column of `student`, where the first attribute in this tuple is identical to the first value in the tuple of `took`, since all occurrences of the same variable in a rule must be assigned the same value. The scope of variables, however, is local to rules, and identically named variables in different rules are considered independent.

The second important benefit of the Datalog formalism is its ability to break up the problem into smaller subproblems, each expressed by simple rules. Thus, complex patterns of computation and logical decisions can be

achieved through rather simple Datalog rules that are stacked one upon another in a rich semantic structure.

For instance, say that in order to take the individual-study course cs298, a junior must have satisfied both requirements. Then we can simply write the following:

Example 8.5 Both requirements must be satisfied to enroll in cs298

$$\texttt{req\_cs298(Name)} \leftarrow \quad \texttt{firstreq(Name)}, \texttt{scndreq(Name)}.$$

Therefore, derived relations can be used as goals in rules in the same fashion as database relations.

Datalog rules discussed so far are nonrecursive rules without negation. Additional expressive power can be achieved by allowing recursion and negation in Datalog. We will next discuss negation; we discuss recursion later in this chapter.

Negation in Datalog rules can only be applied to the goals of the rule. Negation can never be used in the heads of rules. For instance, in Example 8.6, the second goal of the second rule is negated. This rule is meant to compute junior students who did not take course cs143.

Example 8.6 Junior-level students who did not take course cs143

$$\begin{array}{ll} \texttt{hastaken(Name, Course)} \leftarrow & \texttt{took(Name, Course, Grade)}. \\ \texttt{lacks\_cs143(Name)} \leftarrow & \texttt{student(Name, \_, junior)}, \\ & \neg\texttt{hastaken(Name, cs143)}. \end{array}$$

Thus, `hastaken` defines the courses completed by a student, independent of the final grade. Then, the second rule selects those students for whom the pattern `cs143` does not appear in the second column.

A frequent use of negation is in conjunction with universally quantified queries that are often expressed by words such as "each" and "every." For instance, say we would like to express the following query: "find the senior students who completed all requirements for a cs major."

The universally quantified condition "all requirements must be satisfied" can only be expressed in Datalog by transforming it into an equivalent condition where universal quantification is replaced by existential quantification and negation. This transformation normally requires two steps.

The first step is that of formulating the complementary query. For the example at hand, this could be "find students who did not take some of the

courses required for a cs major." This can be expressed using the first rule in Example 8.7. Having derived those senior students who are missing some required courses, as the second step, we can now reexpress the original query as "find the senior students who are NOT missing any requirement for a cs major." This corresponds to the second rule in Example 8.7.

Example 8.7 Find the senior students who completed all the requirements for the cs major: ?`all_req_sat(X)`

```
req_missing(Name) ← student(Name, _, senior),
                    req(cs, Course),
                    ¬hastaken(Name, Course).
all_req_sat(Name) ← student(Name, _, senior),
                    ¬req_missing(Name).
```

Turning a universally quantified query into a doubly negated existential query is never without difficulty, but this is a skill that can be mastered with some practice. Indeed, such a transformation is common in natural languages, particularly in euphemistic nuances. For instance, our last sentence, " ... is never without difficulty," was obtained by rephrasing the original sentence " ... is always difficult."

8.2 Relational Calculi

Relational calculus comes in two main flavors: the domain relational calculus (DRC) and the tuple relational calculus (TRC). The main difference between the two is that in DRC variables denote values of single attributes, while in TRC variables denote whole tuples.

For instance, the DRC expression for a query ?`firstreq(N)` is

$$\{(N) \mid \exists G_1(took(N, cs101, G_1)) \wedge \exists G_2(took(N, cs143, G_2)) \wedge \exists M(student(N, M, junior)) \quad \}$$

The query ?`scndreq(N)` can be expressed as follows:

$$\{(N) \mid \exists G, \exists M(took(N, cs131, G) \wedge G > 3.0 \wedge student(N, M, junior)) \vee \exists G, \exists M(took(N, cs151, G) \wedge G > 3.0 \wedge student(N, M, junior))\}$$

There are obvious syntactic differences that distinguish DRC from Datalog, including the use of set definition by abstraction instead of rules.

Furthermore, DRC formulas contain many additional constructs such as explicit quantifiers, nesting of parentheses, and the mixing of conjunctions and disjunctions in the same formula.

Negation and universal quantification are both allowed in DRC. Therefore, the query ?`all_req_sat(N)` can be expressed either using double negation, or directly using the universal quantifier as shown in Example 8.8. This formula also features the implication sign $\rightarrow$, where $p \rightarrow q$ is just a shorthand for $\neg p \vee q$.

Example 8.8 Using a universal quantifier to find the seniors who completed all cs requirements

$$\{(N) | \quad \exists M(student(N, M, senior)) \wedge \forall C(req(cs, C) \rightarrow \exists G(took(N, C, G))\} \tag{8.1}$$

The additional syntactic complexity of DRC does not produce a more powerful language. In fact, for each domain predicate calculus expression there is an equivalent, nonrecursive Datalog program. The converse is also true, since a nonrecursive Datalog program can be mapped into an equivalent DRC query.

Relational calculus languages are important because they provide a link to commercial database languages. For instance, Query-By-Example (QBE) is a visual query language based on DRC. However, languages such as QUEL and SQL are instead based on TRC.

In TRC, variables range over the tuples of a relation. For instance, the TRC expression for a query ?`firstreq(N)` is the following:

Example 8.9 The TRC equivalent of the query ?`firstreq(N)` in Example 8.3

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

In Example 8.9, t and s are variables denoting, respectively, tuples in `took` and `student`. Thus, $t[1]$ denotes the first component in t (i.e., that corresponding to attribute `Name`); $t[2]$ denotes the `Course` value of this tuple. In general, if $j_1, \ldots, j_n$ denote columns of a relation R, and $t \in R$, then we will use the notation $t[j_1, \ldots, j_n]$ to denote the n-tuple $(t[j_1], \ldots, t[j_n])$.

The main difference between DRC and TRC is that TRC requires an explicit statement of equality, while in DRC equality is denoted implicitly by the presence of the same variable in different places. For instance, in

Example 8.9, the explicit conditions $t[1] = u[1]$ and $s[1] = t[1]$ are needed to express equality joins. Once again, however, these differences do not change the power of the language: TRC and DRC are equivalent, and there are mappings that transform a formula in one language into an equivalent one in the other.

8.3 Relational Algebra

Datalog rules and DRC or TRC formulas are declarative logic-based languages, but relational algebra (RA) is an operator-based language. However, formulas in logical languages can be implemented by transforming them into equivalent RA expressions.

The main operators of relational algebra can be summarized as follows:

1. *Union.* The union of relations R and S, denoted $R \cup S$, is the set of tuples that are in R, or in S, or in both. Thus, it can be defined using TRC as follows:

$$R \cup S = \{t | t \in R \vee t \in S\}$$

   This operation is defined only if R and S have the same number of columns.

2. *Set difference.* The difference of relations R and S, denoted $R - S$, is the set of tuples that belong to R but not to S. Thus, it can be defined as follows: ($t = r$ denotes that both t and r have n components and $t[1] = r[1] \wedge \ldots \wedge t[n] = r[n]$):

$$R - S = \{t | t \in R \wedge \neg \exists r(r \in S \wedge t = r)\}$$

   This operation is defined only if R and S have the same number of columns (arity).

3. *Cartesian product.* The Cartesian product of R and S is denoted $R \times S$.

$$R \times S = \{t | (\exists r \in R)(\exists s \in S)(t[1, \ldots, n] = r \wedge t[n+1, \ldots, n+m] = s)\}$$

   If R has n columns and S has m columns, then $R \times S$ contains all the possible $m + n$ tuples whose first m components form a tuple in R and the last n components form a tuple in S. Thus, $R \times S$ has $m + n$ columns and $|R| \times |S|$ tuples, where $|R|$ and $|S|$ denote the respective cardinalities of the two relations.

4. *Projection.* Let R be a relation with n columns, and $L = \$1, \ldots, \$n$ be a list of the columns of R. Let L' be a sublist of L obtained by (1) eliminating some of the elements, and (2) reordering the remaining ones in an arbitrary order. Then, the projection of R on columns L', denoted $\pi_{L'}$, is defined as follows:

$$\pi_{L'}R \;=\; \{r[L'] \mid r \in R\}$$

5. *Selection.* $\sigma_F R$ denotes the selection on R according to the selection formula F, where F obeys one of the following patterns:

   (i) $\$i\theta C$, where i is a column of R, θ is an arithmetic comparison operator, and C is a constant, or

   (ii) $\$i\theta\$j$, where $\$i$ and $\$j$ are columns of R, and θ is an arithmetic comparison operator, or

   (iii) an expression built from terms such as those described in (i) and (ii), above, and the logical connectives $\vee$, $\wedge$, and $\neg$.

   Then,

   $$\sigma_F R = \{t \mid t \in R \wedge F'\}$$

   where F' denotes the formula obtained from F by replacing $\$i$ and $\$j$ with $t[i]$ and $t[j]$.

   For example, if F is "$\$2 = \$3 \wedge \$1 = bob$", then F' is "$t[2] = t[3] \wedge t[1] = bob$". Thus $\sigma_{\$2=\$3\wedge\$1=bob}R = \{t \mid t \in R \wedge t[2] = t[3] \wedge t[1] = bob\}$.

Additional operators of frequent use that can be derived from these are discussed next.

The *join operator* can be constructed using Cartesian product and selection. In general, a join has the following form: $R \bowtie_F S$, where $F = \$i_1\theta_1\$j_1 \wedge \ldots \wedge i_k\theta_k\$j_k$; $i_1, \ldots, i_k$ are columns of R; $j_1, \ldots, i_k$ are columns of S; and $\theta_1, \ldots, \theta_k$ are comparison operators. Then, if R has arity m, we define $F' = \$i_1\theta_1\$(m + j_1) \wedge \ldots \wedge \$i_k\theta_k\$(m + j_k)$. Therefore,

$$R \bowtie_F S = \sigma_{F'}(R \times S)$$

The *intersection* of two relations can be constructed either by taking the equijoin of the two relations in every column (and then projecting out duplicate columns) or by using the following property: $R \cap S = R - (R - S) = S - (S - R)$.

The *generalized projection* of a relation R is denoted $\pi_L(R)$, where L is a list of column numbers and constants. Unlike ordinary projection, components might appear more than once, and constants as components of the list L are permitted (e.g., $\pi_{\$1,c,\$1}$ is a valid generalized projection). Generalized projection can be derived from the other operators (see Exercise 8.6).

8.4 From Safe Datalog to Relational Algebra

Relational algebra provides a very attractive operational target language onto which the logic-based queries can be mapped. However, only *safe* Datalog programs can be mapped into equivalent relational algebra expressions. From a practical viewpoint, this is hardly a limitation, since enforcing the safety requirement on programs enables the compiler-time detection of rules and queries that are inadequately specified.

For instance, to find grades better than the grade Joe Doe got in cs143, a user might write the following rule:

$$\texttt{bettergrade(G1)} \leftarrow \quad \texttt{took('Joe Doe', cs143, G)}, \texttt{G1} > \texttt{G}.$$

This rule presents the following peculiar traits:

1. *Infinite answers.* Assuming that, say, Joe Doe got the grade of 3.3 (i.e., B+) in course cs143 then there are infinitely many numbers that satisfy the condition of being greater than 3.3.

2. *Lack of domain independence.* A query formula is said to be *domain independent* when its answer only depends on the database and the constants in the query, and not on the domain of interpretation. Clearly, the set of values for `G1` satisfying the rule above depends on what domain we assume for numbers (e.g., integer, rational, or real). Thus, there is no domain independence.

3. *No relational algebra equivalent.* Only database relations are allowed as operands of relational algebra expressions. These relations are finite, and so is the result of every RA expression over these relations. Therefore, there cannot be any RA expression over the database relations that is equivalent to the rule above.

In practical languages, it is desirable to allow only safe formulas, which avoid the problems of infinite answers and loss of domain independence. Unfortunately, the problems of domain independence and finiteness of answers

are undecidable even for nonrecursive queries. Therefore, necessary and sufficient syntactic conditions that characterize safe formulas cannot be given in general. In practice, therefore, we must use sufficient conditions (i.e., conditions that once satisfied ensure domain independence and finiteness of the answers), although such properties might also be satisfied by formulas that do not obey those conditions. A set of simple sufficient conditions for the safety of Datalog programs is presented next.

Definition 8.1 *Safe Datalog. The following is an inductive definition of safety for a program P:*

1. *Safe predicates:* A predicate q of P is *safe* if

   (i) q is a database predicate, or

   (ii) every rule defining q is safe.

2. *Safe variables:* A variable X in rule r is safe if

   (i) X is contained in some *positive goal* $q(t_1, \ldots, t_n)$, where the predicate $q(A_1, \ldots, A_n)$ is safe, or

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

3. *Safe rules:* A rule r is safe if all its variables are safe.

4. The goal $?q(t_1, \ldots, t_n)$ is safe when the predicate $q(A_1, \ldots, A_n)$ is safe.

For every safe Datalog program, there is an equivalent relational algebra expression, generated using the following algorithm:

Algorithm 8.2 *Mapping a safe, nonrecursive Datalog program P into RA*

step 1. P is transformed into an equivalent program P' that does not contain any equality goal by replacing equals with equals and removing the equality goals. For example,

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

is translated into

$$r : \mathtt{s}(\mathtt{Z}, \mathtt{b}, \mathtt{Z}) \leftarrow \quad \mathtt{q}(\mathtt{X}, \mathtt{X}, \mathtt{Y}), \mathtt{p}(\mathtt{Y}, \mathtt{Z}, \mathtt{a}), \mathtt{Z} > 24.3 \ .$$

step 2. The body of a rule r is translated into the RA expression $Body_r$. $Body_r$ consists of the Cartesian product of all the base or derived relations in the body, followed by a selection σ_F, where F is the conjunction of the following conditions: (i) inequality conditions for each such goal (e.g., $Z > 24.3$), (ii) equality conditions between columns containing the same variable, and (iii) equality conditions between a column and the constant occurring in such a column.

For the example at hand, (i) the condition $Z > 24.3$ translates into the selection condition $\$5 > 24.3$, while (ii) the equality between the two occurrences of X translates into $\$1 = \$2$, while the equality between the two Ys maps into $\$3 = \$4$, and (iii) the constant in the last column of `p` maps into $\$6 = a$. Thus we obtain:

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

step 3. Each rule r is translated into a generalized projection on $Body_r$, according to the patterns in the head of r. For the rule at hand, we obtain:

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

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

The mapping just described can be generalized to translate rules with negated goals, as described next. Say that the body of some rule contains a negated goal, such as the following body:

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

Then we consider a *positive body*, that is, one constructed by dropping the negated goal,

$$rp : \ldots \leftarrow \quad \texttt{b1(a,Y)}, \texttt{b2(Y)}.$$

and a *negative* body, that is, one obtained by removing the negation sign from the negated goal,

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

The two bodies so generated are safe and contain no negation, so we can transform them into equivalent relational algebra expressions as per step 2 of Algorithm 8.2; let $Body_{rp}$ and $Body_{rn}$ be the RA expressions so obtained.

Then the body expression to be used in step 3 of said algorithm is simply $Body_r = Body_{rp} - Body_{rn}$.

Finally, observe that by repeating this mapping for each negated goal in the rule we can translate rules with several negated goals. Therefore, we can state the following:

Theorem 8.3 *Let P be a safe Datalog program without recursion or function symbols. Then, for each predicate in P, there exists an equivalent relational algebra expression.*

Safety conditions similar to those described for Datalog can also be generated for DRC and TRC formulas. Thus, safe formulas in each language can be translated into RA, and vice versa (see Exercise 8.9).

The safety conditions used here can be relaxed in several ways to improve the flexibility and ease-of-use of the language. One such extension, discussed in the next chapter, achieves safety by using bindings that are passed in a top-down fashion. Another important extension is to allow the negated goals in a rule to contain existential variables, that is, variables that do not appear anywhere else in the rule. For instance, to express the query "find all senior students that did not take cs143," we might write the first rule in the following example:

Example 8.10 Two equivalent uses of negation

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

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

The first rule in Example 8.10, where `G` does not appear in any positive goal, will be viewed as a convenient shorthand for the other two rules in the same example, which are safe by our previous definition. These last two rules, therefore, define the meaning and the RA equivalent of the first rule.

8.4.1 Commercial Query Languages

Relational query languages represent the results of protracted efforts to simplify the DRC and TRC and make them more user-friendly. For instance, QBE is generally regarded as a very attractive rendering of DRC. Instead, languages such as QUEL and SQL are derived from TRC via simple syntactic modifications. The main modification consists of ensuring that every

tuple variable is *range quantified*; that is, it is explicitly associated with a relation over which it ranges, thus ensuring the safety of the resulting TRC expressions. For instance, the query of Example 8.9 can be rearranged as follows:

Example 8.11 A range-quantified version of Example 8.9

$$\{(t[1]) | \tag{8.2}$$

$$t \in took\ \exists u \in took\ \exists s \in student \tag{8.3}$$

$$(t[2] = cs101 \wedge u[2] = cs143 \wedge t[1] = u[1] \wedge$$
$$s[3] = junior \wedge s[1] = t[1])\} \tag{8.4}$$

Thus, `t` and `u` range over `took`, while `s` ranges over `student`. Then our TRC query consists of three parts: (1) a target-list (8.2), (2) tuple range declaration (8.3), and (3) the conditions (8.4). These are, respectively, mapped into the three basic parts of the SQL query: (1) the **SELECT** part, (2) the **FROM** part, and (3) the **WHERE** part. Also, if $t \in R$, and Name is the name of the j^{th} column of R, then the notation t.Name denotes $t[j]$. Therefore, from Example 8.11, we obtain the SQL query of Example 8.12:

Example 8.12 The SQL translation of Example 8.11

```
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
```

While the need for explicit quantifiers was eliminated in SQL through the use of the **FROM** clause, **EXISTS** and **ALL** are allowed in nested SQL queries. However, the query from Example 8.7 ("find the seniors who satisfied all requirements for a cs degree") cannot be expressed using **ALL**, but must instead be expressed using double negation and existential quantifiers as shown in Example 8.13.

Indeed, while SQL supports the construct **ALL**, various syntactic restrictions limit its applicability to the point that it cannot be used to express universal quantification in many queries, including the current one. In fact, with the exception of set aggregates, the many additional constructs cluttering SQL do not extend the expressive power of the language beyond that of relational algebra or nonrecursive Datalog rules. Therefore, the current

Example 8.13 Find senior students where there is no required cs course our senior has not taken

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

practice in developing database applications is to rely on procedural languages, with embedded SQL subqueries. The interface between the procedural language and SQL must overcome an impedance mismatch, that is, a mismatch between their respective data types and computational paradigms. More powerful query languages are expected to ameliorate these problems because they allow a larger portion of the application to be developed in the database query language. Better data independence and distributed processing are also expected as a result.

In the sections that follow, we will investigate the design of more powerful database query languages, building on Datalog, which, in terms of syntax and semantics, provides a better formal vehicle for such study than SQL and other languages presented so far. This effort will lead to (1) the design of logic-based rule languages similar to Datalog, which have been proven effective in the development of advanced data-intensive applications, and (2) the design of SQL extensions that significantly enhance its power as a relational query language.

8.5 Recursive Rules

Bill of materials (BoM) problems are related to assemblies containing superparts composed of subparts that are eventually composed of basic parts. Consider Example 8.14. The base predicate `assembly(Part, Subpart, Qty)` in the parts database contains parts, their immediate subparts, and the quantity of subparts needed to assemble the part. The base predicate `part_cost(BasicPart, Supplier, Cost, Time)` describes the basic parts,

Example 8.14 Relational tables for a BoM application

part_cost			
BASIC_PART	*SUPPLIER*	*COST*	*TIME*
top_tube	cinelli	20.00	14
top_tube	columbus	15.00	6
down_tube	columbus	10.00	6
head_tube	cinelli	20.00	14
head_tube	columbus	15.00	6
seat_mast	cinelli	20.00	6
seat_mast	cinelli	15.00	14
seat_stay	cinelli	15.00	14
seat_stay	columbus	10.00	6
chain_stay	columbus	10.00	6
fork	cinelli	40.00	14
fork	columbus	30.00	6
spoke	campagnolo	0.60	15
nipple	mavic	0.10	3
hub	campagnolo	31.00	5
hub	suntour	18.00	14
rim	mavic	50.00	3
rim	araya	70.00	1

assembly		
PART	*SUBPART*	*QTY*
bike	frame	1
bike	wheel	2
frame	top_tube	1
frame	down_tube	1
frame	head_tube	1
frame	seat_mast	1
frame	seat_stay	2
frame	chain_stay	2
frame	fork	1
wheel	spoke	36
wheel	nipple	36
wheel	rim	1
wheel	hub	1
wheel	tire	1

that is, parts bought from external suppliers rather than assembled internally. This relation describes the suppliers of each part, and for each supplier the price charged for it and time needed to deliver it.

Assume now that we want to find all the subparts of a given part—not just immediate subparts. Then recursive rules, such as those in Example 8.15, are needed to express this transitive closure query. The second rule in Example 8.15 inductively defines the transitive closure of all subparts; this rule is recursive since the head predicate also appears in the body of the rule.

A nonrecursive rule defining a recursive predicate, such as the first rule in Example 8.15, is called an *exit rule*. Thus, an exit rule provides the base case in the inductive definition of a recursive predicate.

Example 8.15 All subparts: a transitive closure query

```
all_subparts(Part, Sub) ← assembly(Part, Sub, _).
all_subparts(Part, Sub2) ← all_subparts(Part, Sub1),
                           assembly(Sub1, Sub2, _).
```

Once we view assembly(Sub1, Sub2, _) as defining an arc from Sub1 to Sub2, we see that we are basically computing the transitive closure of a graph. Transitive closure computations and its variations are very common in actual applications. Queries involving aggregates are common in BoM applications. For instance, say that a user needs to compute how long it takes to obtain all the basic subparts of an assembly part. (Assuming that the actual assembly time is negligible, this will allow us to estimate how soon an order can be filled.) Then, we can begin with rules that define basic subparts as follows:

Example 8.16 **For each part, basic or otherwise, find its basic subparts (a basic part is a subpart of itself)**

```
basic_subparts(BasicP, BasicP) ← part_cost(BasicP, _, _, _).

basic_subparts(Prt, BasicP) ← assembly(Prt, SubP, _),
                              basic_subparts(SubP, BasicP).
```

Now, we want to find the absolutely shortest time in which we can obtain a basic part, given that the time for delivery might be a function of supplier or even the price charged (fast deliveries might command a premium price). The least-time condition can be expressed using negation, by requiring that there is no faster time for this part.

Example 8.17 **For each basic part, find the least time needed for delivery**

```
fastest(Part, Time) ← part_cost(Part, Sup, Cost, Time),
                      ¬faster(Part, Time).
faster(Part, Time) ← part_cost(Part, Sup, Cost, Time),
                     part_cost(Part, Sup1, Cost1, Time1),
                     Time1 < Time.
```

Then, by combining the last two examples, we can build for each part the list of its basic components, with each component listed with the least time required to get that component.

Example 8.18 **Times required for basic subparts of the given assembly**

```
timeForbasic(AssPart, BasicSub, Time) ←
                        basic_subparts(AssPart, BasicSub),
                        fastest(BasicSub, Time).
```

Thus, the time required to have all the basic parts of a given part is just the longest time required for any such part.

Example 8.19 The maximum time required for basic subparts of the given assembly

```
howsoon(AssPart, Time) ← timeForbasic(AssPart, _, Time),
                         ¬larger(AssPart, Time).
larger(Part, Time) ←     timeForbasic(Part, _, Time),
                         timeForbasic(Part, _, Time1),
                         Time1 > Time.
```

Another family of queries of interest are set aggregates. As seen in Examples 8.17 and 8.19, nonrecursive Datalog with negation can express `min` and `max`. Other aggregates, such as `count` or `sum`, require stratified Datalog with arithmetic. However, counting the elements in a set modulo an integer does not require arithmetic. The program in Example 8.20 determines whether a given base relation `br(X)` contains an even number of elements (i.e., counts the cardinality of this relation *mod* 2). The `next` predicate in the example sorts the elements of `br` into an ascending chain, where the first link of the chain connects the distinguished node `nil` to the least element in `br` (third rule in the example).

Example 8.20 The parity query

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

Observe that Example 8.20 relies on the assumption that the elements of `br` are totally ordered by $>$, and can therefore be visited one at a time using this order.

8.6 Stratification

The predicate dependency graph for a program P is denoted $pdg(P)$ and is defined as follows:

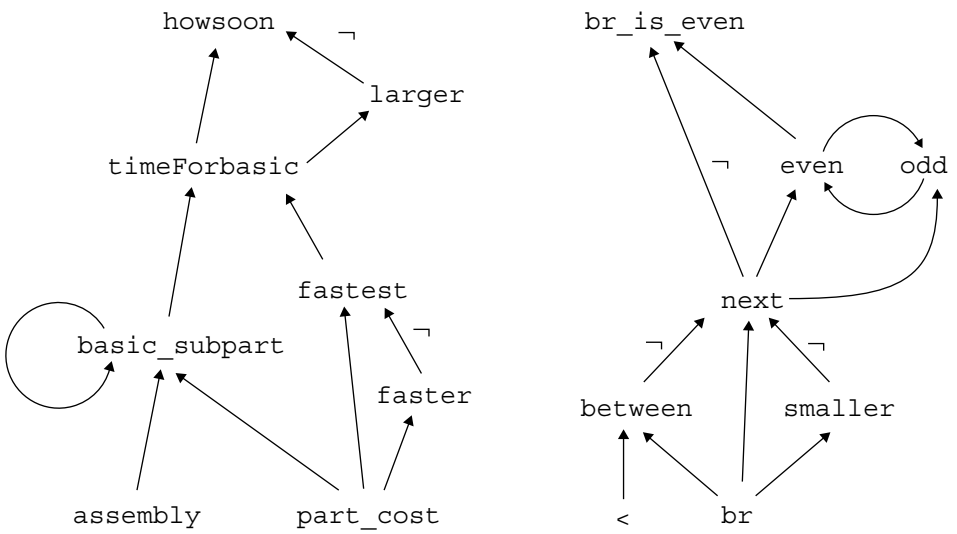

Figure 8.1: Predicate dependency graphs for the BoM query and the parity query

Definition 8.4 *The* predicate dependency graph *for a program P is a graph having as nodes the names of the predicates in P. The graph contains an arc $g \rightarrow b$ if there exists a rule r where g is the name of some goal of r and h is the name of the head of r. If the goal is negated, then the arc is marked as a negative arc.*

The nodes and arcs of the strong components of $pdg(P)$, respectively, identify the recursive predicates and recursive rules of P. Let r be a rule defining a recursive predicate p. The number of goals in r that are mutually recursive with p will be called the *rank* of r.

If $rank(r) = 0$, then r is an exit rule; otherwise, it is a recursive rule. A recursive rule r is called *linear* when $rank(r) = 1$; it is called *nonlinear* when $rank(r) > 1$. Rules with rank 2 and 3 are also called quadratic rules and cubic rules, respectively.

For the BoM program defined by Examples 8.16–8.19, the only recursive predicate is `basic_subparts` and is identified by a loop in the left graph of Figure 8.1. The predicate dependency graph for the parity query program, also shown in Figure 8.1, has a strong component having as nodes the mutually recursive predicates `even` and `odd`.

Observe that in Figure 8.1 no arc marked with negation belongs to a strong component of the graph (a directed cycle). Under this situation the program is said to be *stratifiable*. As discussed in later chapters, programs that are stratifiable have a clear meaning, but programs that are not stratifiable are often ill defined from a semantic viewpoint. In fact, when an arc in a strong component of $pdg(P)$ is marked with negation, then P contains a rule where a predicate p is defined in terms of a goal $\neg p$, or of a goal $\neg q$, where q is mutually recursive with p. Because of the nonmonotonic nature of negation, this might cause contradictions and other semantic problems that will be discussed in more detail later.

Given a stratifiable program P, by applying a topological sorting on $pdg(P)$, the nodes of P can be partitioned into a finite set of n strata $1, \ldots, n$, such that, for each rule $r \in P$, the predicate name of the head of r belongs to a stratum that

(i) is $\geq$ to each stratum containing some positive goal of r, and also

(ii) is strictly $>$ than each stratum containing some negated goal of r.

The strata of P are used to structure the computation so that the predicates of stratum j are used only after every predicate at the lower stratum has been computed. Violations of this rule can produce meaningless results. For our BoM example, for instance, a stratification is shown in Figure 8.1. Observe, for example, that in order to compute `howsoon`, we need to have completed the computation of `larger`, which, in turn, requires the computation of `timeForbasic` and `basic_subpart`. This is a recursive predicate, and many iterations might be needed before its computation is completed. The computation of `howsoon` must wait until such completion; indeed, it is not possible to compute the maximum time required by `basic_subpart` before all basic subparts are known.

A stratification of a program will be called *strict* if every stratum contains either a single predicate or a set of predicates that are mutually recursive.

8.7 Expressive Power and Data Complexity

The *expressive power of a language* L is the set of functions that can be written (programmed) in L. Determining the expressive power of a language and its constructs is fundamental in understanding the limitations of the language and the computational complexity of queries written in such a language.

For instance, transitive closure queries can be expressed using recursive Datalog rules. Moreover, recursive rules are also indispensable for the task, insofar as transitive closures cannot be computed in Datalog or SQL without recursion. Characterizing the expressive power of a query language is often difficult because proving the impossibility of expressing certain classes of queries frequently represents an open research challenge, and much current work is devoted to this topic. This section provides a very short overview of the most basic results on this subject.

A basic requirement for query languages is that they should be able to express all queries expressed by relational algebra. Languages having this property are said to be *relationally complete*. All languages discussed so far meet this requirement, since safe DRC and TRC have the same expressive power as relational algebra, and nonrecursive Datalog with negation is also equivalent to these. The term *first-order* (FO) *languages* is often used to denote the class of these equivalent languages.

At the introduction of the relational model, relational completeness for query languages was viewed as a difficult objective to meet, insofar as many of the early query languages were not relationally complete. Nowadays, commercial languages, such as SQL, are relationally complete, and they also support set aggregate constructs that cannot be expressed by relational algebra but are frequently needed in queries. Thus, the objective of today's advanced database systems is to go well beyond relational completeness. In particular, it is desirable that query languages express every query that can be computed in a polynomial number of steps in the size of the database. These are called *DB-PTIME queries*, and are discussed next.

The notion of *data complexity* is defined by viewing query programs as mappings from the database (the input) to the answer (the output). Thus, complexity measures, such as the big O, are evaluated in terms of the size of the database, which is always finite.[1]

Therefore, a Turing machine is used as the general model of computation, and a database of size n is encoded as a tape of size $O(n)$. Then, all computable functions on the database can be implemented as Turing machines. Some of these machines halt (complete their computation) in a polynomial

[1] Concretely, we might want to measure the database size by the number of characters in the database, or the number of elements in relation columns, or the number of tuples in relations. Assuming that there is a finite number of relations, each with a finite number of columns, and an upper bound to the length of the items in the columns, then these three measures only differ by multiplicative constants of no consequence in normal complexity measures.

number of steps, that is, at most in $O(n^k)$ steps, with k a positive integer; other machines halt in an exponential number of steps; others never halt.

The set of machines that halt in a number of steps that is polynomial in n defines the class of DB-PTIME functions. A language L is said to be in DB-PTIME if every function it computes is polynomial in n. L is said to be *DB-PTIME complete* if L is in DB-PTIME and L can express all DB-PTIME computable functions.

It is easy to prove that FO languages are in DB-PTIME. Actually, sophisticated query optimization strategies and storage structures are used by commercial DBMSs to keep the exponents and coefficients of the polynomials low. But FO languages are not DB-PTIME complete. For instance, they cannot express transitive closures.

As we will show later, recursive Datalog with stratified negation is in DB-PTIME. However, there are still some polynomial queries, such as the parity query, that this language cannot express. The parity query determines whether a set contains an even number of elements. To perform this computation, you need the ability to process the elements in a relation one at a time. Unfortunately, RA operates in the set-at-a-time mode, and Datalog does, too, since its rules can be translated into RA. One way to overcome this limitation is to assume that the elements in the set belong to a totally ordered domain, (i.e., for every two distinct elements, x and y, in the domain, either $x \leq y$ or $y \leq x$). Then the elements of any set can be visited one at a time with the technique illustrated by the parity query (Example 8.20).

In fact, if constants in the database belong to a totally ordered domain, then stratified Datalog can express all polynomial functions, that is, it is DB-PTIME complete.

While assuming that the universe is totally ordered (e.g., in some lexicographical way) is, in itself, not an unreasonable assumption, it leads to a violation of the principle of genericity of queries, which causes the loss of data independence. Therefore, a preferable solution, discussed in later chapters, is that of introducing nondeterministic constructs in Datalog. Nondeterminism can also be used to express more complex queries, including exponential-time queries in Datalog.

Finally, to achieve Turing completeness, and have a language that can express every possible query, an infinite computation domain must be used. In Datalog this can be achieved by using functors.

8.7.1 Functors and Complex Terms

In base relations, functors can be used to store complex terms and variable-length subrecords in tuples. For instance, consider the database of flat assembly parts in Example 8.21. Each planar part is described by its geometric shape and its weight. Different geometric shapes require a different number of parameters. Also `actualkg` is the actual weight of the part, but `unitkg` is the specific weight, from which the actual weight can be derived by multiplying it by the area of the part. For simplicity, we will assume that all parts described are flat in shape and can therefore be described by their planar geometry.

Example 8.21 Flat parts, their number, shape, and weight, following the schema: `part(Part#, Shape, Weight)`

```
part(202, circle(11), actualkg(0.034)).
part(121, rectangle(10, 20), unitkg(2.1)).
part_weight(No, Kilos) ← part(No, _, actualkg(Kilos)).
part_weight(No, Kilos) ← part(No, Shape, unitkg(K)),
                         area(Shape, Area), Kilos = K * Area.

area(circle(Dmtr), A) ←              A = Dmtr * Dmtr * 3.14/4.
area(rectangle(Base, Height), A) ← A = Base * Height.
```

The complex terms `circle(11)`, `actualkg(0.034)`, `rectangle(10, 20)`, and `unitkg(2.1)` are in logical parlance called *functions* because of their syntactic appearance, consisting of a function name (called the *functor*) followed by a list of arguments in parentheses. In actual applications, these complex terms are not used to represent evaluable functions; rather, they are used as variable-length subrecords. Thus, `circle(11)` and `rectangle(10, 20)`, respectively, denote that the shape of our first part is a circle with diameter 11 cm, while the shape of the second part is a rectangle with base 10 cm and height 20 cm. Any number of subarguments is allowed in such complex terms, and these subarguments can, in turn, be complex terms. Thus, objects of arbitrary complexity, including solid objects, can be nested and represented in this fashion.

As illustrated by the first two rules in Example 8.21, functors can also be used as case discriminants to prescribe different computations. The weight of a part is computed in two different ways, depending on whether the third argument of `part` contains the functor `actualkg` or `unitkg`. In the first case, the weight of the part is the actual argument of the functor (e.g., `0.034` for part `202`). In the latter case, the argument gives the specific weight per

cm^2, and its actual weight is computed by multiplying the specific weight by the `Area` of the part. Now, `Area` is derived from `Shape`, according to the computation prescribed by the last two rules, which have been written assuming a top-down passing of parameters from the goal `area(Shape, Area)` to the heads of the rules (the top-down passing of parameters will be further discussed in later sections). Thus, if `Shape` is instantiated to `circle(11)` by the execution of the first goal in the second rule, then the first `area` rule is executed; but if `Shape = rectangle(10, 20)`, then the second rule is executed. This example also illustrates the ability to mix computation with retrieval in a seamless fashion.

The full power of functors comes to life when they are used to generate recursive objects such as lists. Lists can be represented as complex terms having the following form: `list(nil)` (the empty list), and `list(Head, Tail)` for nonempty lists. Given the importance of lists, most logic programming languages provide a special notation for these. Thus, [ ] stands for the empty list, while [`Head`|`Tail`] represents a nonempty list. Then, the notation [`mary`, `mike`, `seattle`] is used as a shorthand for [`mary`, [`mike`, [`seattle`, [ ]]]].

Lists or complex terms are powerful constructs and can be used to write very sophisticated applications; however, they are also needed in basic database applications, such as constructing nested relations from normalized ones. This problem is solved in the following examples:

Example 8.22 A list-based representation for suppliers of `top_tube`

```
part_sup_list(top_tube, [cinelli, columbus, mavic]).
```

Transforming a nested-relation representation into a normalized relation, such as the `ps` relation in Example 8.23, is quite simple.

Example 8.23 Normalizing a nested relation into a flat relation

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

The application of these rules to the facts of Example 8.22 yields

```
ps(top_tube, cinelli)
ps(top_tube, columbus)
ps(top_tube, mavic)
```

However, the inverse transformation (i.e., constructing a nested relation from a normalized relation such as ps above) is not so simple. It requires an approach similar to that used in Example 8.20:

Example 8.24 From a flat relation to a nested one

$$
\begin{array}{ll}
\texttt{between}(\texttt{P},\texttt{X},\texttt{Z}) \leftarrow & \texttt{ps}(\texttt{P},\texttt{X}),\texttt{ps}(\texttt{P},\texttt{Y}),\texttt{ps}(\texttt{P},\texttt{Z}),\texttt{X}<\texttt{Y},\texttt{Y}<\texttt{Z}. \\
\texttt{smaller}(\texttt{P},\texttt{X}) \leftarrow & \texttt{ps}(\texttt{P},\texttt{X}),\texttt{ps}(\texttt{P},\texttt{Y}),\texttt{Y}<\texttt{X}. \\
\texttt{nested}(\texttt{P},[\texttt{X}]) \leftarrow & \texttt{ps}(\texttt{P},\texttt{X}),\neg\texttt{smaller}(\texttt{P},\texttt{X}). \\
\texttt{nested}(\texttt{P},[\texttt{Y}|[\texttt{X}|\texttt{W}]]) \leftarrow & \texttt{nested}(\texttt{P},[\texttt{X}|\texttt{W}]),\texttt{ps}(\texttt{P},\texttt{Y}),\texttt{X}<\texttt{Y}, \\
 & \neg\texttt{between}(\texttt{P},\texttt{X},\texttt{Y}). \\
\texttt{ps\_nested}(\texttt{P},\texttt{W}) \leftarrow & \texttt{nested}(\texttt{P},\texttt{W}),\neg\texttt{nested}(\texttt{P},[\texttt{X}|\texttt{W}]).
\end{array}
$$

Simple extensions of RA are needed to support Datalog with arithmetic and function symbols (see Exercise 8.11).

8.8 Syntax and Semantics of Datalog Languages

Following the general introduction to logic-based query languages in the previous sections, we can now present a more formal definition for such languages. Thus, we relate the syntax and semantics of query languages to those of first-order logic. Then, we introduce the model-theoretic semantics of Datalog programs and present a fixpoint theorem that provides the formal link to the bottom-up implementation of such programs.

8.8.1 Syntax of First-Order Logic and Datalog

First-order logic follows the syntax of context-free languages. Its alphabet consists of the following:

1. *Constants.*
2. *Variables*: In addition to identifiers beginning with uppercase, x, y, and z also represent variables in this section.
3. *Functions*, such as $f(t_1, \ldots, t_n)$, where f is an n-ary functor and $t_1, \ldots, t_n$ are the arguments.
4. *Predicates.*
5. *Connectives*: These include basic logical connectives $\vee$, $\wedge$, $\neg$, and the implication symbols $\leftarrow$, $\rightarrow$, and $\leftrightarrow$.
6. *Quantifiers*: $\exists$ denotes the existential quantifier and $\forall$ denotes the universal quantifier.
7. Parentheses and punctuation symbols, used liberally as needed to avoid ambiguities.

A *term* is defined inductively as follows:

- A variable is a term.
- A constant is a term.
- If f is an n-ary functor and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

Well-formed formulas (WFFs) are defined inductively as follows:

1. If p is an n-ary predicate and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula (called an *atomic formula* or, more simply, an *atom*).
2. If F and G are formulas, then so are $\neg F$, $F \vee G$, $F \wedge G$, $F \leftarrow G$, $F \rightarrow G$, and $F \leftrightarrow G$.
3. 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.

Terms, atoms, and formulas that contain no variables are called *ground*.

Example 8.25 Well-formed formulas in first-order logic

$$\exists G_1(took(N, cs101, G_1)) \wedge \exists G_2(took(N, cs143, G_2)) \wedge \exists M(student(N, M, junior)) \tag{8.5}$$

$$\exists N, \exists M(student(N, M, senior) \wedge \forall C(req(cs, C) \rightarrow \exists G(took(N, C, G))))$$

$$\forall x \forall y \forall z\ (p(x, z) \vee \neg q(x, y) \vee \neg r(y, z)) \tag{8.6}$$

$$\forall x \forall y\ (\neg p(x, y) \vee q(f(x, y), a)) \tag{8.7}$$

A WFF F is said to be a *closed formula* if every variable occurrence in F is quantified. If F contains some variable x that is not quantified, then x is said to be a (quantification-) free variable in F, and F is not a closed formula. The variable N is not quantified in the first formula in Example 8.25 (8.5), so this formula is not closed. The remaining three WFFs in Example 8.25 are closed.

A *clause* is a closed WFF of the form

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

where $A_1, \ldots, A_k, B_1, \ldots, B_n$ are atoms and $x_1, \ldots, x_s$ are all the variables occurring in these atoms. Thus, a clause is the disjunction of

positive and negated atoms, whose every variable is universally quantified. A clause is called a *definite clause* if it contains exactly one positive atom and zero or more negated atoms. Thus a definite clause has the form

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

Since $F \leftarrow G \equiv F \vee \neg G$, the previous clause can be rewritten in the standard rule notation:

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

A is called the *head*, and $B_1, \ \ldots, \ B_n$ is called the *body* of the rule.

In Example 8.25, only the WFFs 8.6 and 8.7 are clauses and are written as follows:

Example 8.26 The rule-based representation of clauses 8.6 and 8.7

$$p(x, z) \leftarrow q(x, y), r(y, z).$$

$$q(f(x, y), a) \leftarrow p(x, y).$$

A definite clause with an empty body is called a *unit clause*. It is customary to use the notation "A." instead of the more precise notation "$A \ \leftarrow$." for such clauses. A *fact* is a unit clause without variables (see Example 8.27).

Example 8.27 A unit clause (everybody loves himself) and three facts

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

Definition 8.5 *A positive logic program is a set of definite clauses.*

We will use the terms *definite clause program* and *positive program* as synonyms.

8.8.2 Semantics

Positive logic programs have a very well defined formal semantics since alternative plausible semantics proposed for these programs have been shown to be equivalent. More general programs (e.g., those containing negation) are less well behaved and more complex in this respect and will be discussed

in later chapters. For the rest of this chapter, the word "program" simply means a positive logic program (i.e., a set of definite clauses).

We will discuss the model-theoretic and fixpoint-based semantics of programs. The former provides a purely declarative meaning to a program, while the latter provides the formal link to the bottom-up implementation of deductive databases. The proof-theoretic approach, which leads to SLD-resolution and top-down execution, will be covered in the next chapter.

8.8.3 Interpretations

The notion of an interpretation for a program P is defined with respect to the constant symbols, function symbols, and predicate symbols that are contained in P. In a slightly more general context, we also define interpretations for any first-order language L, given its set of constant symbols, function symbols, and predicate symbols.

Then the first step for assigning an interpretation to L (and every program written in L) is to select a nonempty set of elements U, called the *universe* (or *domain*) of interpretation. Then, an *interpretation* of L consists of the following:

1. For each constant in L, an assignment of an element in U

2. For each n-ary function in L, the assignment of a mapping from U^n to U

3. For each n-ary predicate q in L, the assignment of a mapping from U^n into true, false (or, equivalently, a relation on U^n)

For definite clause languages and programs, however, it is sufficient to consider Herbrand interpretations, where constants and functions represent themselves. Under Herbrand interpretations, functors are viewed as variable-length subrecords, rather than evaluable functions. Therefore, `rectangle(10, 20)` denotes the actual rectangle rather than a two-place function that computes on its arguments 10 and 20.

Then, the *Herbrand universe* for L, denoted U_L, is defined as the set of all terms that can be recursively constructed by letting the arguments of the functions be constants in L or elements in U_L. (In the case that L has no constants, we add some constant, say, a, to form ground terms.)

Then the *Herbrand base* of L is defined as the set of atoms that can be built by assigning the elements of U_L to the arguments of the predicates.

Therefore, given that the assignment of constants and function symbols is fixed, a *Herbrand interpretation* is defined by assigning 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, $a_1, \ldots, a_n$ denoting elements in U_L. Alternatively, a Herbrand interpretation of L is a subset of the Herbrand base of L.

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

Example 8.28 **Let P be the following program:**

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

In this example, $U_P = \{anne, silvia, marc\}$, and

$$B_P = \{parent(x,y)|x,y \in U_P\} \cup \{father(x,y)|x,y \in U_P\} \cup \{mother(x,y)|x,y \in U_P\} \cup \{anc(x,y)|x,y \in U_P\}$$

Since in Example 8.28 there are four binary predicates, and three possible assignments for the first arguments and three for their second arguments, then $|B_P| = 4 \times 3 \times 3 = 36$. Since there are $2^{|B_P|}$ subsets of B_P, there are 2^{36} Herbrand interpretations for the program in Example 8.28.

Example 8.29 **A program P with an infinite B_P and an infinite number of interpretations**

```
p(f(x)) ←   q(x).
q(a) ←      p(x).
```

Then,

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

where $f^0(a), f^1(a)$, and $f^2(a), \ldots$, stand for $a, f(a)$, and $f((a))$, and so on. Moreover,

$$B_P = \{p(f^n(a)) \mid n \geq 0\} \cup \{q(f^m(a)) \mid m \geq 0\}.$$

8.9 The Models of a Program

Let r be a rule in a program P. Then $ground(r)$ denotes the set of ground instances of r (i.e., all the rules obtained by assigning values from the Herbrand universe U_P to the variables in r).

Example 8.30 Let r be the fourth rule in the Example 8.28

Since there are two variables in said r and $|U_P| = 3$, then $ground(r)$ consists of 3×3 rules:

$$\begin{array}{ll} \texttt{parent(anne, anne)} \leftarrow & \texttt{mother(anne, anne)}. \\ \texttt{parent(anne, marc)} \leftarrow & \texttt{mother(anne, marc)}. \\ \qquad\qquad \ldots & \\ \texttt{parent(silvia, silvia)} \leftarrow & \texttt{mother(silvia, silvia)}. \end{array}$$

The ground version of a program P, denoted $ground(P)$, is the set of the ground instances of its rules:

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

For the program P of Example 8.28, $ground(P)$ contains 9 instantiations of the first rule (since it has two variables and $|U_P| = 3$), 27 instantiations of the second rule, 9 instantiations of the third rule, and 9 of the fourth one, plus the two original facts.

Let I be an interpretation for a program P. Then, every ground atom $a \in I$ is said to be true (or satisfied); if $a \notin I$, then a is said to be false (or not satisfied). Then a formula consisting of ground atoms and logical connectives is defined as true or false (satisfied or not satisfied) according to the rules of propositional logic. Therefore,

Definition 8.6 *Satisfaction. A rule $r \in P$ is said to hold true in interpretation I, or to be satisfied in I, if every instance of r is satisfied in I.*

Definition 8.7 *Model. An interpretation I that makes true all rules P is called a* model *for P.*

Observe that I is a model for P iff it satisfies all the rules in $ground(P)$.

Example 8.31 Interpretations and models for Example 8.28

- If $I_1 = \emptyset$, then every instance of the first and second rules in the example are satisfied since the bodies are always false. I_1, however, is not a model since the third and fourth rules in the example (i.e., the facts) are not satisfied. Thus every interpretation aspiring to be a model must contain every fact in the program.
- Consider now $I_2 = \{mother(anne, silvia), mother(anne, marc)\}$. The first rule is satisfied since the body is always false. However, consider the following instance of the second rule: $\texttt{parent}(\texttt{anne}, \texttt{silvia}) \leftarrow \texttt{mother}(\texttt{anne}, \texttt{silvia})$. Here every goal in the body is satisfied, but the head is not. Thus, I_2 is not a model.
- Now consider $I_3 = \{mother(anne, silvia), mother(anne, marc), parent(anne, silvia), parent(anne, marc), anc(anne, silvia), anc(anne, marc)\}$. This is a model.
- $I_4 = I_3 \cup \{anc(silvia, marc)\}$ is also a model, but it is not a minimal one since it contains a redundant atom.

Lemma 8.8 *Model intersection property. Let P be a positive program, and M_1 and M_2 be two models for P. Then, $M_1 \cap M_2$ is also a model for P.*

Definition 8.9 *Minimal model and least model. A model M for a program P is said to be a* minimal model *for P if there exists no other model M' of P where $M' \subset M$. A model M for a program P is said to be its* least model *if $M' \supseteq M$ for every model M' of P.*

Then, as a result of the last lemma we have the following:

Theorem 8.10 *Every positive program has a least model.*

Proof. Since B_P is a model, P has models, and therefore minimal models. Thus, either P has several minimal models, or it has a unique minimal model, the least model of P. By contradiction, say that M_1 and M_2 are two distinct minimal models, then $M_1 \cap M_2 \subset M_1$ is also a model. This contradicts the assumption that M_1 is a minimal model. Therefore, there cannot be two distinct minimal models for P. □

Definition 8.11 *Let P be a positive program. The least model of P, denoted M_P, defines the meaning of P.*

8.10 Fixpoint-Based Semantics

The least-model semantics provides a logic-based declarative definition of the meaning of a program. We need now to consider constructive semantics and effective means to realize the minimal model semantics. A constructive semantics follows from viewing rules as constructive derivation patterns, whereby, from the tuples that satisfy the patterns specified by the goals in a rule, we construct the corresponding head atoms. As previously discussed, relational algebra can be used to perform such a mapping from the body relations to the head relations. For instance, in Example 8.28, `parent` can be derived through a union operator. Then `grandparent` can be derived from these. However, the recursive predicate `anc` is both the argument and the result of the relational algebra expression. Therefore, we have a *fixpoint equation*, that is, an equation of the form $x = T(x)$, where T is a mapping $U \rightarrow U$. A value of x that satisfies this equation is called a *fixpoint* for T; for an arbitrary T, there might be zero or more fixpoints.

For a positive program P, it is customary to consider the mapping T_P, called the *Immediate consequence operator*, for P, defined as follows:

$$T_P(I) = \{A \in B_P \mid \exists r : A \leftarrow A_1, \ldots, A_n \in ground(P), \{A_1, \ldots, A_n\} \subseteq I\}$$

Thus, T_P is a mapping from Herbrand interpretations of P to Herbrand interpretations of P.

Example 8.32 Let P be the program of Example 8.28

For $I = \{anc(anne, marc), parent(marc, silvia)\}$, we have

$$T_P(I) = \{anc(marc, silvia), anc(anne, silvia), \\ mother(anne, silvia), mother(anne, marc)\}$$

Thus, in addition to the atoms derived from the applicable rules, T_P always returns the database facts and the ground instances of all unit clauses.

Example 8.33 Let P be the program of Example 8.29

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

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

Under the fixpoint semantics, we view a program P as defining the following *fixpoint equation* over Herbrand interpretations:

$$I = T_P(I)$$

In general, a fixpoint equation might have no solution, one solution, or several solutions. However, our fixpoint equation is over Herbrand interpretations, which are subsets of B_P, and thus partially ordered by the set inclusion relationship $\subseteq$. In fact, $(2^{|B_P|}, \subseteq)$ is a partial order (transitive, reflexive, and antisymmetric) and a lattice, where intersection $I_1 \cap I_2$, and union $I_1 \cup I_2$, respectively, define the *lub* and *glb* of the lattice. Furthermore, given a set of elements in 2^{B_P}, there exists the union and intersection of such a set, even if it contains infinitely many elements. Thus, we have a *complete* lattice. Therefore, we only need to observe that T_P for definite clause programs is monotonic (i.e., if $N \leq M$, then $T_P(N) \leq T_P(M)$) to conclude that Knaster/Tarski's theorem applies, yielding the following result:

Theorem 8.12 *Let P be a definite clause program. There always exists a least fixpoint for T_P, denoted $lfp(T_P)$.*

The least-model semantics and the least-fixpoint semantics for positive programs coincide:

Theorem 8.13 *Let P be a definite clause program. Then $M_P = lfp(T_P)$.*

Proof. Let I be a fixpoint for T_P. If I is not a model, then there exists a rule $r \in ground(P)$, where the body of r is satisfied by I but the head $h(r)$ is not in I. Then I cannot be a fixpoint. Thus, every fixpoint is also a model. Vice versa, let M_P be the least model for P. Observe that if $a \in M_P$ and $a \notin T_P(M_P)$, then there is no rule in $ground(P)$ with head a and body satisfied by M_P. Thus, $M_P - \{a\}$ would also be a model—a contradiction. Thus, $T_P(M_P) \supseteq M_P$; but if $T_P(M_P) \supset M_P$, then M_P cannot be a model. Thus, $T_P(M_P) = M_P$. Therefore, M_P is a fixpoint; now, we need to prove that it is the least fixpoint. In fact, $M_P \supset lfp(T_P)$ yields a contradiction, since every fixpoint is a model, and $lfp(T_P)$ would be a smaller model than M_P. Thus, $M_P = lfp(T_P)$. □

In conclusion, the least-fixpoint approach and the least-model approach assign the same meaning to a positive program P: $lfp(T_P) = M_P$.

8.10.1 Operational Semantics: Powers of T_P

For positive programs, $lfp(T_P)$ can simply be computed by repeated applications of T_P. The result of n applications of T_P is called the n^{th} power of T_P, denoted $T_P^{\uparrow n}$, defined as follows:

$$\begin{aligned} T_P^{\uparrow 0}(I) &= I \\ &\cdots \\ T_P^{\uparrow n+1}(I) &= T_P(T_P^{\uparrow n}(I)) \end{aligned}$$

Moreover, with ω denoting the first limit ordinal, we define

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

Of particular interest are the powers of T_P starting from the empty set (i.e., for $I = \emptyset$).

Theorem 8.14 *Let P be a definite clause program. Then $lfp(T_P) = T_P^{\uparrow \omega}(\emptyset)$.*

Proof. To show that $T_P^{\uparrow \omega}(\emptyset) \subseteq T_P(T_P^{\uparrow \omega}(\emptyset))$, let $a \in T_P^{\uparrow \omega}(\emptyset)$. Then, for some integer $k > 0$, $a \in T_P^{\uparrow k}(\emptyset)$. But $T_P^{\uparrow k}(\emptyset) = T_P(T_P^{\uparrow k-1}(\emptyset)) \subseteq T_P(T_P^{\uparrow \omega}(\emptyset))$, since T_P is monotonic and $T_P^{\uparrow k-1}(\emptyset) \subseteq T_P^{\uparrow \omega}(\emptyset)$. Thus, $a \in T_P(T_P^{\uparrow \omega}(\emptyset))$.

Now, to show that $T_P^{\uparrow \omega}(\emptyset) \supseteq T_P(T_P^{\uparrow \omega}(\emptyset))$, observe that every atom in the latter set must be the head of some rule $r \in ground(P)$ whose goals are in $T_P^{\uparrow \omega}(\emptyset)$. Now, observe that $a \in T_P^{\uparrow \omega}(\emptyset)$ iff $a \in T_P^{\uparrow k}(\emptyset)$ for some integer k. Therefore, since r has a finite number of goals, there is an integer k for which all the goals of r are in $T_P^{\uparrow k}(\emptyset)$. Then, for the head of r we have $h(r) \in T_P^{\uparrow k+1}(\emptyset) \subseteq T_P^{\uparrow \omega}(\emptyset)$.

Therefore, we have proven that $T_P^{\uparrow \omega}(\emptyset)$ is a fixpoint for T_P. To prove that it is the least fixpoint, let us show that if $N = T_P(N)$, then $N \supseteq T_P^{\uparrow \omega}(\emptyset)$. Indeed, if $N = T_P(N)$, then $N = T_P^{\uparrow \omega}(N)$. But, since $T_P^{\uparrow \omega}$ is monotonic, $T_P^{\uparrow \omega}(N) \supseteq T_P^{\uparrow \omega}(\emptyset)$. □

The equality $M_P = lfp(T_P) = T_P^{\uparrow \omega}(\emptyset)$ outlines a simple algorithm for computing the least model of a definite clause program. In fact, given that T_P is monotonic, and that $T_P^{\uparrow 0}(\emptyset) \subseteq T_P^{\uparrow 1}(\emptyset)$, by induction it follows that $T_P^{\uparrow n}(\emptyset) \subseteq T_P^{\uparrow n+1}(\emptyset)$. Thus, the successive powers of T_P form an ascending chain. Moreover,

$$T_P^{\uparrow k}\ (\emptyset)\ =\ \bigcup_{n \leq k} T_P^{\uparrow n}\ (\emptyset)$$

Observe that if $T_P^{\uparrow n+1}(\emptyset) = T_P^{\uparrow n}(\emptyset)$, then $T_P^{\uparrow n}(\emptyset) = T_P^{\uparrow \omega}(\emptyset)$. Thus, the least fixpoint and least model can be computed by starting from the bottom and iterating the application of T ad infinitum—or until no new atoms are obtained and the $(n+1)^{th}$ power is identical to the n^{th} power.

Since $T_P^{\uparrow k}$ is also monotonic, for all integers k and for $k = \omega$, we have the following:

Lemma 8.15 *Let P be a definite clause program, with least model M_P. Then, $T_P^{\uparrow \omega}(M) = M_P$ for every $M \subseteq M_P$.*

Proof. Since $\emptyset \subseteq M \subseteq M_P$, then $T_P^{\uparrow \omega}(\emptyset) \subseteq T_P^{\uparrow \omega}(M) \subseteq T_P^{\uparrow \omega}(M_P)$, where $T_P^{\uparrow \omega}(\emptyset) = T_P^{\uparrow \omega}(M_P)$. □

8.11 Bibliographic Notes

The logic-based foundations for the relational data model and query languages were established by E. F. Codd, who introduced the relational calculus and relational algebra and proved their equivalence [115, 116, 117]. Several textbooks, including [437, 2], provide an extensive coverage of these topics.

The topic of logic and databases has been the area of much research work; an incomplete list of published works that explore this area include [182, 183, 308, 355]. A recent survey of the topic can be found in [289]. An interest in logic-based languages for databases was renewed in the 1980s, in part as a result of the emergence of the Prolog language [309, 112, 418]. The design approaches followed by these languages and their supporting deductive database systems are discussed and contrasted with those of Prolog in [296, 465, 302].

The topic of expressive power and computational complexity of query languages is extensively covered in [2]; proofs that transitive closures cannot be expressed in FO languages are presented in [13, 100].

The equivalence of least-fixpoint and least-model semantics for logic programs was proven in [441]; the least-fixpoint theorem for monotonic functions in a lattice is primarily credited to Tarski [428]. Lattices are treated in [60] and in Part V of this volume.

8.12 Exercises

8.1. Using the part_cost relation of Example 8.14, write safe Datalog rules to find those suppliers who supply all basic parts.

8.2. Write a Datalog query to find students who have taken at least two classes, and got the highest grade (possibly with others) in every class they took.

8.3. Express the previous query in SQL using the **EXISTS** construct.

8.4. Universally quantified queries can also be expressed in SQL using the set aggregate **COUNT**. Reformulate the last query and that of Example 8.13 in SQL, using the **COUNT** aggregate.

8.5. A relationally complete relational algebra (RA) contains set union, set difference, set intersection, Cartesian product, selection, and projection.

a. Show that the expressive power of RA does not change if we drop set intersection.

b. List the monotonic operators of RA.

c. Show that there is a loss in expressive power if we drop set difference from the RA described above.

8.6. Define generalized projection using the other RA operators.

8.7. Which of the following rules and predicates are safe if b1 and b2 are database predicates?

$$\begin{aligned} \mathtt{r}_1 : \mathtt{p}(\mathtt{X},\mathtt{X}) \leftarrow \quad & \mathtt{b2}(\mathtt{Y},\mathtt{Y},\mathtt{a}), \mathtt{b1}(\mathtt{X}), \mathtt{X} > \mathtt{Y}. \\ \mathtt{r}_2 : \mathtt{q}(\mathtt{X},\mathtt{Y}) \leftarrow \quad & \mathtt{p}(\mathtt{X},\mathtt{Z}), \mathtt{p}(\mathtt{Z},\mathtt{Y}). \\ \mathtt{r}_3 : \mathtt{s}(\mathtt{X}) \leftarrow \quad & \mathtt{b2}(\mathtt{Y},\mathtt{Y},\mathtt{a}), \mathtt{X} > \mathtt{Y}, \neg \mathtt{b1}(\mathtt{X}). \end{aligned}$$

Translate the safe Datalog rules and predicates in this program into equivalent relational algebra expressions.

8.8. Improve the translation proposed for negated rule goals into RA by minimizing the number of columns in the relations involved in the set difference. Translate the following rule:

$$\mathtt{r}_4 : \mathtt{n}(\mathtt{X},\mathtt{X}) \leftarrow \quad \mathtt{b2}(\mathtt{X},\mathtt{Y},\_), \neg \mathtt{b2}(\mathtt{X},\mathtt{a},\_).$$

8.9. Prove the converse of Theorem 8.3: that is, show that for every RA expression, there exists an equivalent Datalog program, which is safe and nonrecursive.

8.10. Express in Datalog the division operation $R(A, B) \div S(B)$. By translating the rules so obtained into RA, express relation division in terms of the other RA operators.

8.11. Generalize the safety conditions for Datalog to include function symbols. Then extend the RA with the following two operators: the *extended projection*, which extracts the subarguments of a complex term, and the *combine* operator (denoted by γ), which builds complex terms from simple tokens. For instance, if a relation R only contains the tuple ('Jones', degree_year(ba, 1996)), then $S = \pi_{\$1,\$1.0,\$1.1}R$ contains the tuple ('Jones', degree_year, ba). Then $\gamma_{\texttt{degree}(\$3)}\texttt{S}$ returns the tuple ('Jones', degree(ba)). Prove that every safe nonrecursive Datalog program can be mapped into extended RA algebra expressions.

8.12. The following employee relation, `emp(Eno, Ename, MgNo)`, specifies the name and manager number of each employee in the company. The management structure in this company is a strict hierarchy. Write transitive closure rules to construct `emp_all_mgrs(Eno, AllMangrs)`, where `AllMangrs` is a list containing the employee numbers for all managers of `Eno` in the management chain.

8.13. Write a Datalog program to compute how many courses required for a cs degree each senior cs student is missing.

8.14. Prove the model intersection property.

8.15. Show that for each positive program P,

$$T_p^{\uparrow n+1}(\emptyset) \supseteq T_p^{\uparrow n}(\emptyset)$$

8.16. Let P be a positive Datalog program. Show that $T_P^{\uparrow\omega}$ can be computed in time that is polynomial in the size of P's Herbrand base.

Chapter 9

Implementation of Rules and Recursion

9.1 Operational Semantics: Bottom-Up Execution

In the last chapter, we saw that the least model for a program P can be computed as $T_P^{\uparrow k}(\emptyset)$, where k is the lowest integer for which $T_P^{\uparrow k}(\emptyset) = T_P^{\uparrow k+1}(\emptyset)$, if such an integer exists, or the first ordinal ω otherwise. The need to compute to the first ordinal can only occur in the presence of an infinite Herbrand universe (e.g., when there are function symbols). For basic Datalog, without function symbols or arithmetic, the universe is finite, and the computation of the least model of a program ends after a finite number of steps k.

This property forms the basis for the bottom-up computation methods used by deductive databases. Several improvements and tuning for special cases are needed, however, to make this computation efficient. This chapter describes these improved bottom-up execution techniques and other execution techniques that are akin to top-down execution strategies used in Prolog.

9.2 Stratified Programs and Iterated Fixpoint

In most systems, $T_P^{\uparrow\omega}(\emptyset)$ is computed by strata. Unless otherwise specified, we will use strict stratifications in the discussion that follows.

The stratified computation is *inflationary*, in the sense that the results of the previous iterations on T_P are kept and augmented with the results of

the new iteration step. This yields the concept of the inflationary immediate consequence operator.

Definition 9.1 *Let P be program. The* inflationary immediate consequence operator *for P, denoted Υ_P, is a mapping on subsets of B_P defined as follows:*

$$\Upsilon_P(I) = T_P(I) \cup I$$

It is easy to prove by induction that $\Upsilon_P^{\uparrow n}(\emptyset) = T_P^{\uparrow n}(\emptyset)$. (The computation $T_P^{\uparrow\omega}(\emptyset)$ is frequently called *inflationary fixpoint* computation.) Thus, we have the following properties:

$$M_P = lfp(T_P) = T_P^{\uparrow\omega}(\emptyset) = lfp(\Upsilon_P) = \Upsilon_P^{\uparrow\omega}(\emptyset)$$

Algorithm 9.2 *Iterated fixpoint for program P stratified in n strata. Let P_j, $1 \leq j \leq n$, denote the rules with their head in the j^{th} stratum.*

Then, M_j is inductively constructed as follows:

1. $M_0 = \emptyset$ *and*
2. $M_j = \Upsilon_{P_j}^{\uparrow\omega}(M_{j-1})$.

Theorem 9.3 *Let P be a positive program stratified in n strata, and let M_n be the result produced by the iterated fixpoint computation. Then, $M_P = M_n$, where M_P is the least model of P.*

Proof. We want to prove that, for all $1 \leq j \leq n$, $M_j = \Upsilon_{\leq j}^{\uparrow\omega}(M_{j-1})$, where $\Upsilon_{\leq j}$ denotes the inflationary immediate consequence operator for the rules up to the j^{th} stratum. The property is trivial for $j = 1$. Now assume that it holds for every M_i, where $0 \leq i \leq j-1$. Now, the equality $\Upsilon_{P_j}^{\uparrow k}(M_{j-1}) = \Upsilon_{\leq j}^{\uparrow k}(M_{j-1})$ holds for $k = 0$, and we will next prove that if it holds for k, then it also holds for $k+1$. Indeed, then $\Upsilon_{\leq j}^{\uparrow k+1}(M_{j-1}) = \Upsilon_{\leq j}(\Upsilon_{\leq j}^{\uparrow k}(M_{j-1})) = \Upsilon_{\leq j}(\Upsilon_{P_j}^{\uparrow k}(M_{j-1}))$. But every rule with its head in a stratum $< j$ produces atoms that are already in M_{j-1}. Thus, $\Upsilon_{\leq j}(\Upsilon_{P_j}^{\uparrow k}(M_{j-1})) = \Upsilon_{P_j}(\Upsilon_{P_j}^{\uparrow k}(M_{j-1}))$, and $M_j = \Upsilon_{\leq j}^{\uparrow\omega}(M_{j-1})$.

Therefore, by using double induction over the n strata, we conclude that $M_n = \Upsilon_{\leq n}^{\uparrow\omega}(M_{n-1})$. But $M_n = \Upsilon_{\leq n}^{\uparrow\omega}(M_{n-1}) = \Upsilon_P^{\uparrow\omega}(M_{n-1})$. But since $\emptyset \subseteq M_{n-1} \subseteq M_P$, then $\Upsilon_P^{\uparrow\omega}(M_{n-1}) = M_P$, by Lemma 8.15. □

The iterated fixpoint computation defined by Algorithm 9.2 terminates whenever the original fixpoint computation terminates. However, when this

requires an infinite number of steps, then the iterated fixpoint will not go past the first stratum, requiring infinitely many iterations. In this case, we have a transfinite computation that is more accurately described by the term "procedure," than by "algorithm."

9.3 Differential Fixpoint Computation

The algorithm for computing $\Upsilon_{P_j}^{\uparrow\omega}(M_j)$ for stratum j of the iterated fixpoint computation is given by Algorithm 9.4, where for notational expediency, we let Υ stand for Υ_{P_j} and M stand for M_{j-1}.

Algorithm 9.4 *The inflationary fixpoint computation for each stratum*

$S := M;$
$S' := \Upsilon(M)$
while $(S \subset S')$
{
    $S := S';$
    $S' := \Upsilon(S)$
}

This computation can be improved by observing that $T(M) = T_E(M)$ and $\Upsilon(M) = \Upsilon_E(M)$, where T_E denotes the immediate consequence operator for the exit rules and Υ_E denotes its inflationary version. Conversely, let T_R denote the immediate consequence operator for the recursive rules and let Υ_R be its inflationary version. Then, $\Upsilon(S)$ in the **while** loop can be replaced by $\Upsilon_R(S)$, while $\Upsilon(M)$ outside the loop can be replaced by $\Upsilon_E(M)$. For instance, for the linear ancestor rules of Example 9.1, Υ_E and Υ_R are defined by rules r_1 and r_2, respectively.

Example 9.1 Left-linear ancestor rules

r_1 : anc(X, Y) ← parent(X, Y).
r_2 : anc(X, Z) ← anc(X, Y), parent(Y, Z).

Even after this first improvement, there is still significant redundancy in the computation. In fact, before entering the loop, the set S contains the pairs parent/person. After the first iteration, this contains the pairs grandparent/person, along with the old pairs parent/person. Thus, at the second iteration, the recursive rule produces the pairs great-grandparent/person, but also produces the old pairs great-grandparent/person from previous iterations. In general, the j^{th} iteration step also recomputes all atoms obtained

in the $j-1^{th}$ step. This redundancy can be eliminated by using finite differences techniques, which trace the derivations over two steps. Let us use the following notation:

1. S is a relation containing the atoms obtained up to step $j-1$.
2. $S' = \Upsilon_R(S)$ is a relation containing the atoms produced up to step j.
3. $\delta S = \Upsilon_R(S) - S = T_R(S) - S$ denotes the atoms newly obtained at step j (i.e., the atoms that were not in S at step $j-1$).
4. $\delta' S = \Upsilon_R(S') - S' = T_R(S') - S'$ are the atoms obtained at step j.

We can now rewrite Algorithm 9.4 as follows:

Algorithm 9.5 *Differential fixpoint*

$S := M;$
$\delta S := \Upsilon_E(M);$
$S' := \delta S \cup S;$
while $(\delta S \neq \emptyset)$
{
    $\delta' S := T_R(S') - S';$
    $S := S'$;
    $\delta S := \delta' S;$
    $S' := S \cup \delta S$
}

For Example 9.1, let anc, δanc, and anc$'$, respectively, denote ancestor atoms that are in S, δS, and $S' = S \cup \delta S$. Then, when computing $\delta S' := T_R(S') - S'$ in Algorithm 9.5, we can use a T_R defined by the following rule:

$$\delta'\texttt{anc}(\texttt{X},\texttt{Z}) \leftarrow \quad \texttt{anc}'(\texttt{X},\texttt{Y}), \texttt{parent}(\texttt{Y},\texttt{Z}).$$

Now, we can split anc$'$ into anc and δanc, and rewrite the last rule into the following pairs:

$$\delta'\texttt{anc}(\texttt{X},\texttt{Z}) \leftarrow \quad \delta\texttt{anc}(\texttt{X},\texttt{Y}),\ \texttt{parent}(\texttt{Y},\texttt{Z}).$$
$$\delta'\texttt{anc}(\texttt{X},\texttt{Z}) \leftarrow \quad \texttt{anc}(\texttt{X},\texttt{Y}),\ \texttt{parent}(\texttt{Y},\texttt{Z}).$$

The second rule can now be eliminated, since it produces only atoms that were already contained in anc$'$ (i.e., in the S' computed in the previous iteration). Thus, in Algorithm 9.5, rather than using $\delta' S := T_R(S') - S'$, we

can write $\delta' S := T_R(\delta S) - S'$ to express the fact that the argument of T_R is the set of delta tuples from the previous step, rather the set of all tuples obtained so far. This transformation holds for all linear recursive rules.

Consider now a quadratic rule (i.e., one where the recursive predicate appears twice in the body). Say, for instance, that we have the following:

Example 9.2 Quadratic ancestor rules

$$\begin{aligned} \texttt{ancs}(\texttt{X},\texttt{Y}) &\leftarrow \texttt{parent}(\texttt{X},\texttt{Y}). \\ \texttt{ancs}(\texttt{X},\texttt{Z}) &\leftarrow \texttt{ancs}(\texttt{X},\texttt{Y}), \texttt{ancs}(\texttt{Y},\texttt{Z}). \end{aligned}$$

The recursive rule can be transformed for Algorithm 9.5 as follows:

$$r : \delta'\texttt{ancs}(\texttt{X},\texttt{Z}) \leftarrow \texttt{ancs}'(\texttt{X},\texttt{Y}), \texttt{ancs}'(\texttt{Y},\texttt{Z}).$$

By partitioning the relation corresponding to the first goal into an ancs part and a δancs part, we obtain:

$$\begin{aligned} r_1 : \delta'\texttt{ancs}(\texttt{X},\texttt{Z}) &\leftarrow \delta\texttt{ancs}(\texttt{X},\texttt{Y}), \texttt{ancs}'(\texttt{Y},\texttt{Z}). \\ r_2 : \delta'\texttt{ancs}(\texttt{X},\texttt{Z}) &\leftarrow \texttt{ancs}(\texttt{X},\texttt{Y}),\ \texttt{ancs}'(\texttt{Y},\texttt{Z}). \end{aligned}$$

Now, by the same operation on the second goal, we can rewrite the second rule just obtained into

$$\begin{aligned} r_{2,1} : \delta'\texttt{ancs}(\texttt{X},\texttt{Z}) &\leftarrow \texttt{ancs}(\texttt{X},\texttt{Y}), \delta\texttt{ancs}(\texttt{Y},\texttt{Z}). \\ r_{2,2} : \delta'\texttt{ancs}(\texttt{X},\texttt{Z}) &\leftarrow \texttt{ancs}(\texttt{X},\texttt{Y}),\ \texttt{ancs}(\texttt{Y},\texttt{Z}). \end{aligned}$$

Rule $r_{2,2}$ produces only "old" values, and can thus be eliminated; we are then left with rules r_1 and $r_{2,1}$, below:

$$\begin{aligned} \delta'\texttt{ancs}(\texttt{X},\texttt{Z}) &\leftarrow \delta\texttt{ancs}(\texttt{X},\texttt{Y}), \texttt{ancs}'(\texttt{Y},\texttt{Z}). \\ \delta'\texttt{ancs}(\texttt{X},\texttt{Z}) &\leftarrow \texttt{ancs}(\texttt{X},\texttt{Y}),\ \delta\texttt{ancs}(\texttt{Y},\texttt{Z}). \end{aligned}$$

Thus, for nonlinear rules, the immediate consequence operator used in Algorithm 9.5 has the more general form $\delta' S := T_R(\delta S, S, S') - S'$, where $\delta S = S' - S$.

Observe that even if S and S' are not totally eliminated, the resulting computation is usually much more efficient, since it is typically the case that $n = |\delta S| \ll N = |S| \approx |S'|$. The original ancs rule, for instance, requires the equijoin of two relations of size N; after the differentiation we need to compute two equijoins, each joining a relation of size n with one of size N.

Observe also that there is a simple analogy between the symbolic differentiation of these rules and the chain rule used to differentiate expressions in calculus. For instance, the body of the left-linear anc rule of Example 9.1

consists of the conjunction of a recursive predicate `anc` (a variable predicate) followed by a constant predicate. Thus, we have a pattern $x \cdot c$, where x stands for a variable and c for a constant. Therefore, $\delta(x \cdot c) = (\delta x) \cdot c$. For the quadratic rule defining `ancs` in Example 9.2, we have instead the pattern $\delta(x \cdot y) = \delta x \cdot y + \delta y \cdot x$, where the symbol '+' should be interpreted as set union.

The general expression of $T_R(\delta S, S, S')$ for a recursive rule of rank k is as follows. Let

$$r : Q_0 \leftarrow c_0, Q_1, c_1, Q_2, \ldots, Q_k, c_k$$

be a recursive rule, where $Q_1, \ldots, Q_k$ stand for occurrences of predicates that are mutually recursive with Q_0, and $c_1, \ldots, c_k$ stand for the remaining goals. Then, r is symbolically differentiated into k rules as follows:

$$\begin{array}{lllll}
r_1 : \delta' Q_0 \leftarrow & c_0, \ \delta Q_1, \ c_1, \ Q'_2, & \ldots & Q'_k, \ c_k \\
r_2 : \delta' Q_0 \leftarrow & c_0, \ Q_1, \ c_1, \ \delta Q_2, & \ldots & Q'_k, \ c_k \\
 & & \ldots & \\
r_j : \delta' Q_0 \leftarrow & \ldots & \delta Q_j & Q'_k, \ c_k \\
 & & \ldots & \\
r_k : \delta' Q_0 \leftarrow & c_0, \ Q_1, \ c_1, \ Q_2, & \ldots & \delta Q_k, \ c_k
\end{array}$$

Observe the pattern of delta predicates on the main diagonal, unprimed predicates to the left of the diagonal, and primed predicates to its right. This result was obtained by expanding each goal Q'_j into δQ_j and Q_j for ascending js. If we perform the expansion by descending js, the deltas become aligned along the other diagonal.

In terms of implementation, the differential fixpoint is therefore best realized by rewriting the original rules. Alternatively you can first transform the original rules into relational algebra expressions and apply symbolic differentiation to those.

9.4 Top-Down Execution

In the top-down procedural semantics of logic programs, each goal in a rule body is viewed as a call to a procedure defined by other rules in the same stratum or in lower strata. Consider, for instance, the rules from Example 9.3. The goal `area(Shape, Area)` in rule r_2 can be viewed as a call to the procedure `area` defined by rules r_3 and r_4. At the time when the procedure is called, `Shape` is instantiated to values such as `circle(11)` or

Example 9.3 The rules of Example 8.21

r_1 : part_weight(No, Kilos) ← part(No, _, actualkg(Kilos)).
r_2 : part_weight(No, Kilos) ← part(No, Shape, unitkg(K)), area(Shape, Area), Kilos = K * Area.

r_3 : area(circle(Dmtr), A) ← A = Dmtr * Dmtr * 3.14/4.
r_4 : area(rectangle(Base, Height), A) ← A = Base * Height.

rectangle(10, 20) by the execution of r_3 and r_4 goals. The argument Area is instead assigned a value by the execution of the two called area rules. Thus, A and Area can be viewed as what in procedural languages are respectively called formal parameters and actual parameters. Unlike procedural languages, however, the arguments here can be complex, and the passing of parameters is performed through a process known as *unification*. For instance, if Shape = rectangle(10, 20), this will be made equal to (unified to) the first argument of the second area rule, rectangle(Base, Height), by setting Base = 10 and Height = 20.

9.4.1 Unification

Definition 9.6 *A* substitution *θ is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each v_i is a distinct variable, and each t_i is a term distinct from v_i. Each t_i is called a* binding *for v_i. The substitution θ is called a* ground substitution *if every t_i is a ground term.*

Let E be a term and θ a substitution for the variables of E. Then, $E\theta$ denotes the result of applying the substitution θ to E (i.e., of replacing each variable with its respective binding). For instance, if $E = p(x, y, f(a))$ and $\theta = \{x/b, y/x\}$, then $E\theta = p(b, x, f(a))$. If $\gamma = \{x/c\}$, then $E\gamma = p(c, y, f(a))$. Thus, variables that are not part of the substitution are left unchanged.

Definition 9.7 *Let $\theta = \{u_1/s_1, \ldots, u_m/s_m\}$ and $\delta = \{v_1/t_1, \ldots, v_n/t_n\}$ be substitutions. Then the* composition *$\theta\delta$ of θ and δ is the substitution obtained from the set*

$$\{u_1/s_1\delta, \ldots, u_m/s_m\delta, v_1/t_1, \ldots, v_n/t_n\}$$

by deleting any binding $u_i/s_i\delta$ for which $u_i = s_i\delta$ and deleting any binding v_j/t_j for which $v_j \in \{u_1, \ldots, u_m\}$.

For example, let $\theta = \{(x/f(y), y/z)\}$ and $\delta = \{x/a, y/b, z/y\}$. Then $\theta\delta = \{x/f(b), z/y\}$.

Definition 9.8 *A substitution θ is called a* unifier *for two terms A and B if $A\theta = B\theta$.*

Definition 9.9 *A unifier θ for these two terms is called a* most general unifier *(mgu), if for each other unifier γ, there exists a substitution δ such that $\gamma = \theta\delta$.*

For example, the two terms $p(f(x), a)$ and $p(y, f(w))$ are not unifiable because the second arguments cannot be unified.

The two terms $p(f(x), z)$ and $p(y, a)$ are unifiable, since $\delta = \{y/f(a), x/a, z/a)\}$ is a unifier. A most general unifier is $\theta = \{y/f(x), z/a\}$. Note that $\delta = \theta\{x/a\}$.

There exist efficient algorithms to perform unification; such algorithms either return a most general unifier or report that none exists.

9.4.2 SLD-Resolution

Consider a rule $r : A \leftarrow B_1, \ldots, B_n$, and a query goal $\leftarrow g$, where r and g have no variables in common (we can rename the variables of r to new distinct names to satisfy this requirement). If there exists an mgu δ that unifies A and g, then the *resolvent* of r and g is the goal list:

$$\leftarrow B_1\delta, \ldots, B_n\delta.$$

The following abstract algorithm describes the top-down proof process for a given program P and a goal list $\mathbf{G}$.

Algorithm 9.10 *SLD-resolution*

Input: *A first-order program P and a goal list* $\mathbf{G}$.
Output: *Either an instance* $\mathbf{G}\delta$ *that was proved from P, or* failure.

begin *Set* Res = $\mathbf{G}$;
    While Res *is not empty, repeat the following:*
        Choose a goal g from Res;
        Choose a rule $A \leftarrow B_1, \ldots, B_n (n \geq 0)$ *from P*
            such that A and g unify under the mgu δ,
            (renaming the variables in the rule as needed);

If no such rule exists, then
output failure *and exit;*
else Delete g from Res*;*
Add $B_1, \ldots, B_n$ *to* Res*;*
Apply δ *to* Res *and* $\mathbf{G}$*;*
If Res *is empty, then output* $\mathbf{G}\delta$
end

Example 9.4 SLD-resolution on a program with unit clauses

$$\begin{aligned}&\mathtt{s(X,Y)} \leftarrow \quad \mathtt{p(X,Y)}, \mathtt{q(Y)}.\\ &\mathtt{p(X,3)}.\\ &\mathtt{q(3)}.\\ &\mathtt{q(4)}.\end{aligned}$$

Let us consider the top-down evaluation of the goal ?$\mathtt{s(5,X)}$ as applied to this program:

1. The initial goal list is
$$\leftarrow \mathtt{s(5,W)}$$
2. We choose $\mathtt{s(5,W)}$ as our goal and find that it unifies with the first rule under the substitution $\{X/5, Y/W\}$. The new goal list is
$$\leftarrow \mathtt{p(5,W)}, \mathtt{q(W)}$$
3. This time, say that we choose $\mathtt{q(W)}$ as a goal and find that it unifies with the fact $\mathtt{q(3)}$, under the substitution $\{\mathtt{W/3}\}$. (This goal also unifies with $\mathtt{q(4)}$, under the substitution $\{\mathtt{W/4}\}$, but say that we choose the first substitution). Then, the new goal list is
$$\leftarrow \mathtt{p(5,3)}$$
which unifies with the fact $\mathtt{p(X,3)}$ under the substitution $\{\mathtt{X/5}\}$. At this point, the goal list becomes empty and we report success. Thus, a top-down evaluation returns the answer $\{\mathtt{W/3}\}$ for the query $\leftarrow \mathtt{s(5,W)}$ from the example program.

However, if in the last step above, we choose instead the fact $\mathtt{q(4)}$, then under substitution $\{\mathtt{W/4}\}$, we obtain the following goal list:
$$\leftarrow \mathtt{p(5,4)}$$
This new goal cannot unify with the head of any rule; thus SLD-resolution returns failure.

Therefore, at each step, SLD-resolution chooses nondeterministically

- a next goal from the goal list, and
- a next rule from those whose head unifies with the goal just selected.

Therefore, a single instance of an SLD-resolution can return either success or failure, depending on the choices made. However, say q is a predicate without bound arguments. Then we can consider all possible choices and collect the results returned by successful instances of SLD-resolution starting from the goal $\leftarrow \mathtt{q}$. This is known as the *success set* for q; the union of the success sets for all the predicates in program P is equal to the least model of P. This ensures the theoretical equivalence of top-down semantics and bottom-up semantics.

The generation of the success set for a given predicate is possible (e.g., using breadth-first search); however, this is considered too inefficient for most practical applications. Thus, systems such as Prolog use depth-first exploration of alternatives, whereby goals are always chosen in a left-to-right order and the heads of the rules are also considered in the order they appear in the program. Then, the programmer is given responsibility for ordering the rules and their goals in such a fashion as to guide Prolog into successful and efficient searches. The programmer must also make sure that the procedure never falls into an infinite loop. For instance, say that our previous ancestor example is written as follows:

$$\begin{aligned} \mathtt{anc(X,Z)} &\leftarrow \mathtt{anc(X,Y), parent(Y,Z).} \\ \mathtt{anc(X,Z)} &\leftarrow \mathtt{parent(X,Y).} \end{aligned}$$

and that our goal is ?anc(marc, mary). By computing the resolvent of this goal with the head of the first rule, we obtain

$$?\mathtt{anc(marc, Y_1), parent(Y_1, mary).}$$

By replacing the first goal with its resolvent with the head of the first rule, we obtain the following goals:

$$?\mathtt{anc(marc, Y_2), parent(Y_2, Y_1), parent(Y_1, mary).}$$

An additional resolution of the first goal with the first rule yields

$$?\mathtt{anc(marc, Y_3), anc(Y_3, Y_2), parent(Y_2, Y_1), parent(Y_1, mary).}$$

Thus at each step we construct a longer and longer list, without ever returning a single result. When working with Prolog, however, the programmer is aware of the fact that the goals are visited from left to right, and

the rule heads are searched in the same order as they are written. Thus, the programmer will list the rules and order the goals in the rules so as to avoid infinite loops. In our case, this means the exit rule is placed before the recursive one, and the `parent` goal is placed before the recursive `anc` goal, yielding the following:

Example 9.5 Computing ancestors using Prolog

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

In many cases, however, rule reordering does not ensure safety from infinite loops. Take, for instance, the nonlinear version of ancestor:

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

This will first produce all the ancestors pairs; then it will enter a perpetual loop. (This is best seen by assuming that there is no `parent` fact. Then, the recurring failure of the first rule forces the second rule to call itself in an infinite loop.)

Even when the rules are properly written, as in Example 9.5, directed cycles in the underlying parent relation will cause infinite loops. In the particular case of our parent relation, cycles are not expected, although they might result from homonyms and corrupted data. But the same rules could also be used to compute the transitive closures of arbitrary graphs, which normally contain cycles.

Therefore, the bottom-up computation works in various situations where the top-down approach flounders in infinite loops. Indeed, the bottom-up operational semantics is normally more robust than the top-down semantics. However, the top-down approach is superior in certain respects—particularly with its ability to take advantage of constants and constraints that are part of the query goals to reduce the search space. This can be illustrated by the following example:

Example 9.6 Blood relations

```
anc(Old, Young) ←        parent(Old, Young).
anc(Old, Young) ←        anc(Old, Mid), parent(Mid, Young).
grandma(Old, Young) ←    parent(Mid, Young), mother(Old, Mid).
parent(F, Cf) ←          father(F, Cf).
parent(M, Cm) ←          mother(M, Cm).
```

In a query such as ?grandma(GM, marc), marc unifies with Young, which in turn unifies first with Cf and then with Cm. This results in the father relation being searched for tuples where the second component is marc—an efficient search if an index is available on this second column. If this search yields, say, tom, then this value is also passed to Mid, which is instantiated to tom. Thus, the goal mother(Old, tom) is now solved, and if, say, ann is the mother of tom, then the value GM = ann is returned. For the sake of discussion, say that several names are found when searching for the father of marc; then each of these names is passed to the goal mother, and new answers are generated for each new name (assuming that Prolog is in an all-answers mode). When no more such names are found, then the substitution M = marc is attempted, and the second parent rule is processed in similar fashion.

The passing of constants from the calling goals to the defining rules can also be used in the execution of some recursive predicates. For instance, say that in Example 9.6, we have a query ?anc(milton, SV). Then, milton must unify with the first argument in anc, through zero or more levels of recursive calls, until, via the exit rule, it is passed to the first argument of parent, and from there to the base relations.

Advanced deductive database systems mix the basic bottom-up and top-down techniques to combine their strengths. Some systems adopt Prolog's SLD-resolution extended with memorization to overcome various problems, such as those created by cycles in the database. Many systems keep the bottom-up, fixpoint-based computation, but then use special methods to achieve a top-down propagation of query constants similar to that described in this section. The next sections describe these methods, beginning with techniques used for nonrecursive predicates, and proceeding to those used for recursive predicates of increasing complexity.

9.5 Rule-Rewriting Methods

The grandma predicate can be computed using the following relational algebra expression:

$$GRANDMA = \pi_{\$3,\$2}((FATHER \cup MOTHER) \bowtie_{\$1=\$2} MOTHER)$$

which is the result of replacing selections on Cartesian products with equivalent joins in the RA expression produced by Algorithm 8.2. Then the answer to the query goal ?grandma(marc, GM) is $\sigma_{\$2=marc}\ GRANDMA$. But this approach is inefficient since it generates all pairs grandma/grand-child, even if most of them are later discarded by the selection $\sigma_{\$2=marc}$. A better

approach is to transform the original RA expression by pushing selection into the expression as is currently done by query optimizers in relational databases. Then we obtain the equivalent expression:

$$\pi_{\$3,\$2}((\sigma_{\$2=marc}FATHER \cup \sigma_{\$2=marc}MOTHER) \bowtie_{\$1=\$2} MOTHER)$$

In the RA expression so obtained, only the parents of `marc` are selected from the base relations `mother` and `father` and processed through the rest of the expression. Moreover, since this selection produces a binary relation where all the entries in the second column are equal to `marc`, the projection $\pi_{\$1}$ could also be pushed into the expression along with selection.

The optimization performed here on relational algebra can be performed directly by specializing the original rules via an SLD-like pushing of the query constants downward (i.e., into the rules defining the goal predicate); this produces the following program, where we use the notation `X/a` to denote that X has been instantiated to `a`:

Example 9.7 Find the grandma of `marc`

```
?grandma(GM, marc)
grandma(Old, Young/marc) ← parent(Mid, Young/marc),
                           mother(Old, Mid).
parent(F, Cf/marc) ←       father(F, Cf/marc).
parent(M, Cm/marc) ←       mother(M, Cm/marc).
```

Thus, the second argument in the predicate `parent` is set equal to the constant `marc`.

9.5.1 Left-Linear and Right-Linear Recursion

If, in Example 9.6, we need to compute all the `anc` pairs, then a bottom-up approach provides a very effective computation for this recursive predicate. However, consider the situation where the goal contains some constant; for example, say that we have the query ?`anc(tom, Desc)`. As in the case of nonrecursive rules, we want to avoid the wasteful approach of generating all the possible ancestor/person pairs, later to discard all those whose first component is not `tom`. For the recursive `anc` rule of Example 9.6, we can observe that the value of `Old` in the head is identical to that in the body; thus we can specialize our recursive predicate to `anc(tom, _)` throughout the fixpoint computation. Therefore, the `anc` rules can be specialized into those

of Example 9.8, where constant tom has been pushed into the recursive predicate.[1]

Example 9.8 The descendants of tom

```
?anc(tom, Desc).
anc(Old/tom, Young) ← parent(Old/tom, Young).
anc(Old/tom, Young) ← anc(Old/tom, Mid), parent(Mid, Young).
```

As previously discussed, Prolog performs this operation during execution. Most deductive databases prefer a compilation-oriented approach where the program is compiled for a *query form*, such as anc($Name, X). The dollar sign before Name denotes that this is a deferred constant, i.e., a parameter whose value will be given at execution time. Therefore, deferred constants are treated as a constant by the compiler, and the program of Example 9.8 is rewritten using $Name as the first argument of anc.

Transitive-closure-like computations can be expressed in several equivalent formulations; the simplest of these use recursive rules that are either left-linear or right-linear. The left-linear version of anc is that of Example 9.6. Consider now the right-linear formulation of ancestor:

Example 9.9 Right-linear rules for the descendants of tom

```
anc(Old, Young) ← parent(Old, Young).
anc(Old, Young) ← parent(Old, Mid), anc(Mid, Young).
```

With the right-linear rules of Example 9.9, the query ?anc($Name, X) can no longer be implemented by specializing the rules. (To prove that, say that we replace Old with the constant $Name = tom; then, the transitive closure cannot be computed using parent(tom, Mid), which only yields children of tom, while the grandchildren of tom and their children are also needed.)

While it is not possible to specialize the program of Example 9.9 for a query ?anc($Name, X), it is possible to transform it into an equivalent program for which such a specialization will work. Take, for instance, the right-linear program of Example 9.9; this can be transformed into the equivalent left-linear program of Example 9.6, on which the specialization approach can then be applied successfully. While recognizing the equivalence of programs is generally undecidable, many simple left-linear rules can be detected and

[1] As a further improvement, the constant first argument might also be dropped from the recursive predicate.

transformed into their equivalent right-linear counterparts. Symmetric conclusions follow for the left-linear program of Example 9.6, which, for a query such as ?anc(Y, $D), is transformed into its right-linear equivalent of Example 9.9. Techniques for perfoming such transformations will be discussed in Section 9.6.

After specialization, left-linear and right-linear rules can be be supported efficiently using a single fixpoint computation. However, more complex recursive rules require more sophisticated methods to exploit bindings in query goals. As we shall see in the next section, these methods generate a pair of fixpoint computations.

9.5.2 Magic Sets Method

To illustrate the basic idea behind magic sets, let us first consider the following example, consisting of two nonrecursive rules that return the names and addresses of senior students:

Example 9.10 **Find the graduating seniors and the addresses of their parents**

```
snr_par_add(SN, PN, Paddr) ← senior(SN), parent(PN, SN),
                             address(PN, Paddr).
senior(SN) ←                 student(SN, _, senior), graduating(SN).
```

A bottom-up computation on the rules of Example 9.10 determines graduating seniors, their parents, and the parents' addresses in an efficient manner. But, say that we need to find the address of a particular parent, for example, the address of Mr. Joe Doe, who has just called complaining that he did not get his invitation to his daughter's graduation. Then, we might have the following query: ?snr_par_add(SN,'Joe Doe', Addr). For this query, the first rule in Example 9.10 can be specialized by letting PN ='Joe Doe'. Yet, using a strict bottom-up execution, the second rule still generates all names of graduating seniors and passes them up to the senior(SN) of the first rule. An optimization technique to overcome this problem uses an auxiliary "magic" relation computed as follows:

Example 9.11 **Find the children of Joe Doe, provided that they are graduating seniors**

```
snr_par_add_q('Joe Doe').
m.senior(SN) ← snr_par_add_q(PN), parent(PN, SN).
```

The fact snr_par_add_q('Joe Doe') stores the bound argument of the original query goal. This bound argument is used to compute a value of SN that is then passed to m.senior(SN) by the bottom-up rule in Example 9.11, emulating what the first rule in Example 9.10 would do in a top-down computation. We can now improve the second rule of Example 9.10 as follows:

Example 9.12 Restricting search via magic sets

```
senior(SN) ← m.senior(SN),
             student(SN, _, senior), graduating(SN).
```

Therefore, the bottom-up rules of Example 9.12 are designed to emulate the top-down computation where the binding is passed from SN in the head to the first goal of parent. This results in the instantiation of SN, which is then passed to the argument of senior.

The "magic sets" notion is very important for those recursive predicates that are not amenable to the specialization treatment used for left-linear and right-linear rules. For instance, the recursive rule in Example 9.13 is a linear rule that is neither left-linear nor right-linear.

Example 9.13 People are of the same generation if their parents are of the same generation

```
?sg(marc, Who).
sg(X, Y) ← parent(XP, X), sg(XP, YP), parent(YP, Y).
sg(A, A).
```

The recursive rule here states that X and Y are of the same generation if their respective parents XP and YP also are of the same generation. The exit rule sg(X, X) states that every element of the universe is of the same generation as itself. Obviously this rule is unsafe, and we cannot start a bottom-up computation from it. However, consider a top-down computation on these rules, assuming for simplicity that the fact parent(tom, marc) is in the database. Then, the resolvent of the query goal with the first rule is ← parent(XP, marc), sg(XP, YP), parent(YP, Y). Then, by unifying the first goal in this list with the fact parent(tom, marc), the new goal list becomes ← sg(tom, YP), parent(YP, Y). Thus, the binding was passed from the first argument in the head to the first argument of the recursive predicate in the body. Now, the recursive call unfolds as in the previous case, yielding the parents of tom, who are the grandparents of marc. In summary, the top-down computation generates all the ancestors of marc using the recursive rule. This computation causes the instantiation of variables X and XP,

while Y and YP remain unbound. The basic idea of magic sets is to emulate this top-down binding passing using rules to be executed in a bottom-up fashion. Therefore, we can begin by restricting our attention to the bound arguments and use the following rule: $\mathtt{sg(X)} \leftarrow \mathtt{parent(XP, X)}, \mathtt{sg(XP)}$. Then, we observe that the top-down process where bindings are passed from X to XP through parent can be emulated by the bottom-up execution of the magic rule $\mathtt{m.sg(XP)} \leftarrow \mathtt{m.sg(X)}, \mathtt{parent(XP, X)}$; the rule is constructed from the last one by exchanging the head with the recursive goal (and adding the prefix "m."). Finally, as the exit rule for the magic predicate, we add the fact m.sg(marc), where marc is the query constant.

In summary, the magic predicate m.sg is computed as shown by the first two rules in Example 9.14. Example 9.14 also shows how the original rules are rewritten with the addition of the magic goal m.sg to restrict the bottom-up computation.

Example 9.14 The magic sets method applied to Example 9.13

$\mathtt{m.sg(marc)}.$
$\mathtt{m.sg(XP)} \leftarrow \quad \mathtt{m.sg(X)}, \mathtt{parent(XP, X)}.$
$\mathtt{sg'(X, X)} \leftarrow \quad \mathtt{m.sg(X)}.$
$\mathtt{sg'(X, Y)} \leftarrow \quad \mathtt{parent(XP, X)}, \mathtt{sg'(XP, YP)}, \mathtt{parent(YP, Y)}, \mathtt{m.sg(X)}.$
$?\mathtt{sg'(marc, Z)}.$

Observe that, in Example 9.14, the exit rule has become safe as a result of the magic sets rewriting, since only people who are ancestors of marc are considered by the transformed rules. Moreover, the magic goal in the recursive rule is useful in narrowing the search because it eliminates people who are not ancestors of marc.

Following our strict stratification approach, the fixpoint for the magic predicates will be computed before that of the modified rules. Thus, the magic sets method can be viewed as an emulation of the top-down computation through a cascade of two fixpoints, where each fixpoint is then computed efficiently using the differential fixpoint computation.

The fixpoint computation works well even when the graph representing parent is a directed acyclic graph (DAG) or contains directed cycles. In the case of a DAG, the same node and its successors are visited several times using SLD-resolution. This duplication is avoided by the fixpoint computation, since every new result is compared against those previously memorized. In the presence of directed cycles, SLD-resolution flounders in an infinite loop, while the magic sets method still works.

An additional virtue for the magic sets method is its robustness, since the method works well in the presence of multiple recursive rules and even nonlinear rules (provided that the binding passing property discussed in Section 9.6 holds).

One problem with the magic sets method is that the computation performed during the first fixpoint might be repeated during the second fixpoint. For the example at hand, for instance, the ancestors of `marc` are computed during the computation of `ms.sg` and revisited again as descendants of those ancestors in the computation of `sg'`. The counting method and the supplementary magic sets technique discussed next address this problem.

9.5.3 The Counting Method

The task of finding people who are of the same generation as `marc` can be reexpressed as that of finding the ancestors of `marc` and their levels, where `marc` is a zero-level ancestor of himself, his parents are first-generation (i.e., first-level) ancestors, his grandparents are second-generation ancestors, and so on. This computation is performed by the predicate `sg_up` in Example 9.15:

Example 9.15 Find ancestors of marc, and then their descendants

```
sg_up(0, marc).
sg_up(J + 1, XP) ←   parent(XP, X), sg_up(J, X).
sg_dwn(J, X) ←       sg_up(J, X).
dwn(J − 1, Y) ←      sg_dwn(J, YP), parent(YP, Y).
?dwn(0, Z).
```

Here, we have used `sg_up` to replace the top-down computation in the original example; thus, `sg_up` computes the ancestors of `marc` while increasing the level of ancestors. Then, the original exit rule (every person is of the same generation as himself) was used to switch to the computation of descendants. This computation is performed by `sg_dwn`, which also decreases the level counter at each application of the recursive rule. Once we return to level zero, we have a person who is of the same generation as `marc`, as per the modified query goal in Example 9.15. Here we have two fixpoint computations, where the second fixpoint does not duplicate the first computation, except for reversing the original counting of levels. It should be understood that, while the meaning of our same-generation example helps us to recognize the equivalence between the original program and the rewritten program, this equivalence nevertheless holds for all programs that obey the same patterns of bound arguments. As discussed in later sections, an

analysis of patterns of bound arguments that occur during a top-down computation is performed by the compiler to decide the applicability of methods such as magic sets or counting, and to implement these methods by rewriting the original rules into modified rules that yield the same query results.

The counting method mimics the original top-down SLD-resolution to such an extent that it also shares some of its limitations. In particular, cycles in the database will throw the rewritten rules into a perpetual loop; in fact, if $\mathtt{sg\_up}(\mathtt{J}, \mathtt{XP})$ is true and XP is a node in the loop, then $\mathtt{sg\_up}(\mathtt{J} + \mathtt{K}, \mathtt{XP})$, with K the length of the cycle, holds as well.

Another problem with counting is its limited robustness, since for more complex programs, the technique becomes inapplicable or requires several modifications. For instance, let us revise Example 9.13 by adding the goal $\mathtt{XP} \neq \mathtt{YP}$ to the recursive rule, to avoid the repeated derivation of people who are of the same generation as themselves. Then, the rules defining sg_up must be modified to memorize the values of XP, since these are needed in the second fixpoint. By contrast, the supplementary magic technique discussed next disregards the level information and instead relies on the systematic memorization of results from the first fixpoint, to avoid repeating the same computation during the second fixpoint.

9.5.4 Supplementary Magic Sets

In addition to the magic predicates, supplementary predicates are used to store the pairs bound-arguments-in-head/bound-arguments-in-recursive-goal produced during the first fixpoint. For instance, in Example 9.16, we compute spm.sg, which is then used during the second fixpoint computation, since the join of spm.sg with $\mathtt{sg}'$ in the recursive rule returns the memorized value of X for each new XP.

Example 9.16 The supplementary magic method applied to Example 9.13

$$
\begin{array}{ll}
\mathtt{m.sg}(\mathtt{marc}). & \\
\mathtt{m.sg}(\mathtt{XP}) \leftarrow & \mathtt{m.sg}(\mathtt{X}), \mathtt{parent}(\mathtt{XP}, \mathtt{X}). \\
\mathtt{spm.sg}(\mathtt{X}, \mathtt{XP}) \leftarrow & \mathtt{parent}(\mathtt{XP}, \mathtt{X}), \mathtt{m.sg}(\mathtt{X}). \\
\mathtt{sg}'(\mathtt{X}, \mathtt{X}) \leftarrow & \mathtt{m.sg}(\mathtt{X}). \\
\mathtt{sg}'(\mathtt{X}, \mathtt{Y}) \leftarrow & \mathtt{sg}'(\mathtt{XP}, \mathtt{YP}), \mathtt{spm.sg}(\mathtt{X}, \mathtt{XP}), \mathtt{parent}(\mathtt{YP}, \mathtt{Y}). \\
?\mathtt{sg}'(\mathtt{marc}, \mathtt{Z}). &
\end{array}
$$

The supplementary magic rules used in Example 9.16 are in a form that illustrates that this is a refinement of the basic magic sets method previously

described, and in fact the two terms are often used as synonyms. Frequently, the magic predicate and the supplementary magic predicate are written in a mutually recursive form. Thus, for Example 9.16, we have the following rules:

Example 9.17 The magic and supplementary magic rules for 9.13

```
m.sg(marc).
spm.sg(X, XP) ←   m.sg(X), parent(XP, X)
m.sg(XP) ←        spm.sg(X, XP).
```

To better understand how the method works, let us revise the previous example. Say that we only want to search up to k^{th} generations where the parents and their children lived in the same state. Then, we obtain the following program:

Example 9.18 People who are of the same generation through common ancestors who are less than 12 levels remote and always lived in the same state

```
?stsg(marc, 12, Z).
stsg(X, K, Y) ←      parent(XP, X), K > 0, KP = K − 1,
                     born(X, St), born(XP, St),
                     stsg(XP, KP, YP),
                     parent(YP, Y).
stsg(X, K, X).
```

Given that the first two arguments of `stsg` are bound, the supplementary magic method yields:

Example 9.19 The supplementary magic method for Example 9.18

```
m.stsg(marc, 12).
spm.stsg(X, K, XP, KP) ←  m.stsg(X, K),
                          parent(XP, X), K > 0, KP = K − 1,
                          born(X, St), born(XP, St).
m.stsg(X, K) ←            spm.stsg(X, K, XP, KP).

stsg(X, K, X)             m.stsg(X, K).
stsg(X, K, Y) ←           stsg(XP, KP, YP), spm.stsg(X, K, XP, KP),
                          parent(YP, Y).
```

As illustrated by this example, not all the bound arguments are memorized. Only those that are needed for the second fixpoint are stored in the supplementary magic relations. In our case, for instance, `St` is not included.

Because of its generality and robustness, the supplementary magic technique is often the method of choice in deductive databases. In fact, the method works well even when there are cycles in the underlying database. Moreover, the method entails more flexibility with arithmetic predicates. For instance, the expression $\texttt{KP} = \texttt{K} - 1$ is evaluated during the first fixpoint, where `K` is given and the pair $(\texttt{K}, \texttt{KP})$ is then memorized in the supplementary relations for use in the second fixpoint. However, with the basic magic sets method from the second fixpoint, `K` can only be computed from the values of `KP` taken from $\delta\texttt{stsg}(\texttt{XP}, \texttt{KP}, \texttt{YP})$, provided that the equation $\texttt{KP} = \texttt{K} - 1$ is first solved for `K`. Since this is a simple equation, solving it is a simple task for a compiler; however, solving more general equations might either be very difficult or impossible. An alternative approach consists in using the arithmetic equality as is, by taking each value of `K` from the magic set and computing $\texttt{K} - 1$. However, this computation would then be repeated with no change at each step of the second fixpoint computation. The use of supplementary magic predicates solves this problem in a uniform and general way since the pairs $\texttt{K}, \texttt{KP}$ are stored during the first fixpoint and then used during the second fixpoint.

The supplementary magic method can be further generalized to deal with nonlinear rules, including nonlinear rules as discussed in the next section (see also Exercise 9.7).

9.6 Compilation and Optimization

Most deductive database systems combine bottom-up techniques with top-down execution. Take for instance the flat parts program shown in Example 9.3, and say that we want to print a list of part numbers followed by their weights using the following query: ?`part_weight(Part,Weight)`. An execution plan for this query is displayed by the *rule-goal graph* of Figure 9.1.

The graph depicts a top-down, left-to-right execution, where all the possible unifications with rule heads are explored for each goal. The graph shows the names of the predicates with their bound/free *adornments* positioned as superscripts. Adornments are vectors of f or b characters. Thus, a k^{th} character in the vector being equal to b or f denotes that the k^{th} argument in the predicate is respectively bound or free. An argument in a predicate is said to be *bound* when all its variables are instantiated; otherwise the argument is said to be *free*, and denoted by f.

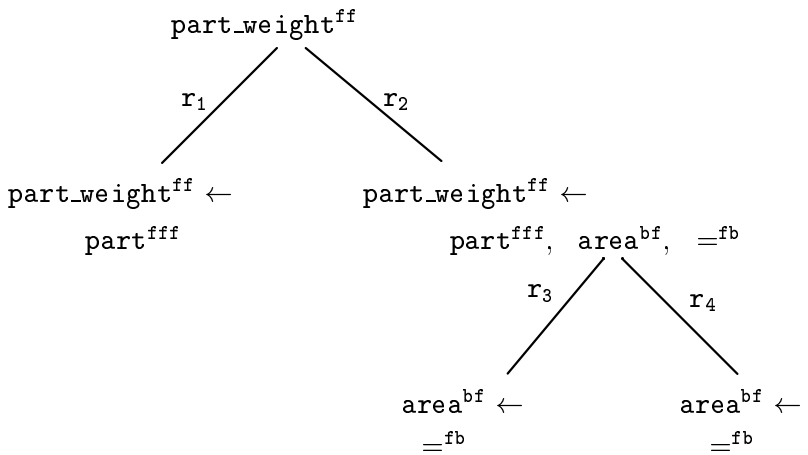

Figure 9.1: The rule-goal graph for Example 9.3

9.6.1 Nonrecursive Programs

The rule-goal graph for a program P is denoted $rgg(P)$. The rule-goal graph for a nonrecursive program is constructed as follows:

Algorithm 9.11 *Construction of the rule-goal graph $rgg(P)$ for a nonrecursive program P.*

1. *Initial step:* The query goal is adorned according to the constants and deferred constants (i.e., the variables preceded by \$), and becomes the root of $rgg(P)$.

2. *Bindings passing from goals to rule heads:* If the calling goal g unifies with the head of the rule r, with mgu γ, then we draw an edge (labeled with the name of the rule, i.e., r) from the adorned calling goal to the adorned head, where the adornments for $h(r)$ are computed as follows: (i) all arguments bound in g are marked bound in $h(r)\gamma$; (ii) all variables in such arguments are also marked bound; and (iii) the arguments in $h(r)\gamma$ that contain only constants or variables marked bound in (ii) are adorned b, while the others are adorned f.

   For instance, say that our goal is $g = p(f(X_1), Y_1, Z_1, a)$, and the head of r is $h(r) = p(X_2, g(X_2, Y_2), Y_2, W_2)$. (If g and $h(r)$ had variables in common, then a renaming step would be required here.) A most general unifier exists for the two: $\gamma = \{X_2/f(X_1),\ Y_1/g(f(X_1), Y_2),\ Z_1/Y_2, W_2/a\}$; thus, bindings might be passed from this goal to this

head in a top-down execution, and the resulting adornments of the head must be computed.

The unified head is $h(r)\gamma = p(f(X_1), g(f(X_1), Y_2), Y_2, a)$. For instance, say that the goal was adorned p^{bfffb}; then variables in the first argument of the head (i.e., X_1) are bound. The resulting adorned head is p^{bfffb}, and there is an edge from p^{bfffb} to p^{bfffb} $\leftarrow$. But if the adorned goal is p^{fbfbb}, then all the variables in the second argument of the head (i.e., X_1, Y_2) are bound. Then the remaining arguments of the head are bound as well. In this case, there is an edge from the adorned goal p^{fbfbb} to the adorned head p^{bbbbb} $\leftarrow$.

3. *Left-to-right passing of bindings to goals:* A variable X is bound after the n^{th} goal in a rule, if X is among the bound head variables (as for the last step), or if X appears in one of the goals of the rule preceding the n^{th} goal.

   The $(n+1)^{th}$ goal of the rule is adorned on the basis of the variables that are bound after the n^{th} goal.

For simplicity of discussion, we assume that the rule-goal graph for a nonrecursive program is a tree, such as that of Figure 9.1. Therefore, rather than drawing multiple edges from different goals to the same adorned rule head, we will duplicate the rule head to ensure that a tree is produced, rather than a DAG.

The rule-goal graph determines the safety of the execution in a top-down mode and yields an overall execution plan, under the simplifying assumption that the execution of a goal binds all the variables in the goal. The safety of the given program (including the bound query goal) follows from the fact that certain adorned predicates are known to be safe a priori.

For instance, base predicates are safe for every adornment. Thus, $\texttt{part}^{\texttt{fff}}$ is safe. Equality and comparison predicates are treated as binary predicates. The pattern $\theta^{\texttt{bb}}$ is safe for θ denoting any comparison operator, such as $\leq$ or $>$. Moreover, there is the special case of $=^{\texttt{bf}}$ or $=^{\texttt{fb}}$ where the free argument consists of only one variable; in either case the arithmetic expression in the bound argument can be computed and the resulting value can be assigned to the free variable.[2]

Then, we have the following definition of safety for a program whose rule-goal graph is a tree:

[2]These represent basic cases that can be treated by any compiler. As previously indicated, a sophisticated compiler could treat an expression, such as $2 * X + 7 = 35$, as safe, if rewriting it as $X = (35 - 7)/2$ is within the capabilities of the compiler.

Definition 9.12 *Let P be a program with rule-goal graph $rgg(P)$, where $rgg(P)$ is a tree. Then P is safe if the following two conditions hold:*
(i) Every leaf node of $rgg(P)$ is safe a priori, and
(ii) every variable in every rule in $rgg(P)$ is bound after the last goal.

Given a safe $rgg(P)$, there is a simple execution plan to compute rules and predicates in the program. Basically, every goal with bound adornments generates two computation phases. In the first phase, the bound values of a goal's arguments are passed to its defining rules (its children in the rule-goal graph). In the second phase, the goal receives the values of the f-adorned arguments from its children. Only the second computation takes place for goals without bound arguments. Observe that the computation of the heads of the rules follows the computation of all the goals in the body. Thus, we have a strict stratification where predicates are computed according to the postorder traversal of the rule-goal graph.

Both phases of the computation can be performed by a relational algebra expression. For instance, the set of all instances of the bound arguments can be collected in a relation and passed down to base relations, possibly using the magic sets technique—resulting in the computation of semijoins against the base relations. In many implementations, however, each instance of bound arguments is passed down, one at a time, to the children, and then the computed values for the free arguments are streamed back to the goal.

9.6.2 Recursive Predicates

The treatment of recursive predicates is somewhat more complex because a choice of recursive methods must be performed along with the binding passing analysis.

The simplest case occurs when the goal calling a recursive predicate has no bound argument. In this case, the recursive predicate, say p, and all the predicates that are mutually recursive with it, will be computed in a single differential fixpoint. Then, we fall back into the treatment of rules for the nonrecursive case, where

1. step 3 of Algorithm 9.11 is performed assuming that rule heads have no bound argument
2. safety analysis is performed by treating the recursive goals (i.e., p and predicates mutually recursive with it) as safe a priori—in fact, they are bound to the values computed in the previous step.

When the calling goal has some bound arguments, then, a *binding passing analysis* is performed to decide which method should be used for the case at

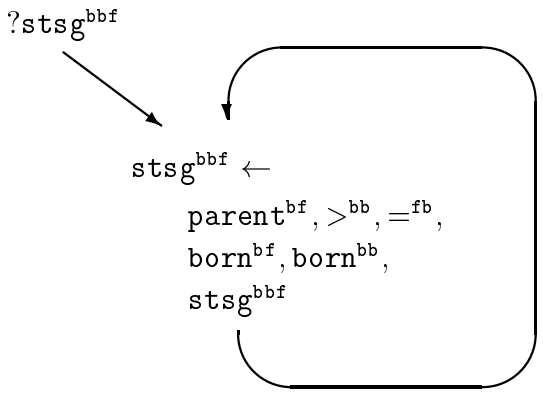

Figure 9.2: Binding passing analysis for the program of Example 9.19

hand. After this analysis, the program is rewritten according to the method selected.

Figure 9.2 illustrates how the binding passing analysis is performed on recursive rules. The binding passing from a goal to the recursive rule heads remains the same as that used for the nonrecursive case (step 2 in Algorithm 9.11). There are, however, two important differences. The first is that we allow cycles in the graph, to close the loop from a calling recursive goal to a matching adorned head already in the graph. The second difference is that the left-to-right binding passing analysis for recursive rules is more restrictive than that used at step 3 of Algorithm 9.11; only particular goals (called chain goals) can be used.

An adorned goal q^γ in a recursive rule r is called a *chain goal* when it satisfies the following conditions:

1. *SIP independence of recursive goals:* q is not a recursive goal (i.e., not the same predicate as that in the head of r, nor a predicate mutually recursive with q; however, recursive predicates of lower strata can be used as chain goals).

2. *Selectivity:* q^γ has some argument bound (according to the bound variables in the head of r and the chain goals to the left of q^γ).

3. *Safety:* q^γ is a safe goal.[3]

[3]If q is not a recursive predicate, then safety is defined above. If q is a recursive goal, then it belongs to a lower stratum; therefore, safety can be determined independently of the safety of q.

The basic idea behind the notion of chain goals is that the binding in the head will have to reduce the search space. Any goal that is called with all its adornment free will not be beneficial in that respect. Also, there is no sideway information passing (SIP) between two recursive goals; bindings come only from the head through nonrecursive goals.

The algorithm for adorning the recursive predicates and rules constructs a set of adorned goals **A** starting from the initial query goal (or a calling goal) q that has adornment γ, where γ contains some bound argument.

Algorithm 9.13 *Binding passing analysis for recursive predicates*

1. *Initially* $\mathbf{A} = \{q^\gamma\}$, *with* q^γ *the initial goal, where* q *is a recursive predicate and* γ *is not a totally free adornment.*
2. *For each* $h \in \mathbf{A}$, *pass the binding to the heads of rules defining* q.
3. *For each recursive rule, determine the adornments of its recursive goals (i.e., of* q *or predicates mutually recursive with* q*).*
4. *If the last step generated adornments not currently in* **A**, *add them to* **A** *and resume from step 2. Otherwise halt.*

The calling goal g is said to have the *binding passing property* when **A** does not contain any recursive predicate with totally free adornment. In this case, we say that g has the *unique* binding passing property when **A** contains only one adornment for each recursive predicate.

When the binding passing property does not hold, then a totally free adornment occurs, and mutually recursive predicates must be computed as if the calling goal had no bound arguments. Otherwise, the methods described in the previous sections are applicable, and the recursive program is rewritten according to the method selected.

9.6.3 Selecting a Method for Recursion

For simplicity of discussion, we assume that the unique binding passing property holds and concentrate on the rewriting for the magic sets method, which can then be used as the basis for other methods.

Let $q^\gamma \in \mathbf{A}$, and r be a recursive rule defining q. Then, if the recursive rank of r is k, then there are k magic rules corresponding to r: one for each recursive goal in r. If p is one of these goals, then the head of the magic rule is named $m.p$, and has as arguments the arguments of p bound according to q^γ. The body of the magic rule consists of the following goals: the recursive goal $m.q$ with the bound arguments in q^γ, and the chain goals of r.

The (one and only) exit rule for all the magic predicates is actually the fact $m.g'$, where g' is obtained from the calling goal by eliminating its free arguments.

Finally, each original rule r is augmented with the addition of a magic goal as follows. Say that q is the head of r, $q^\gamma \in \mathbf{A}$, and q' is obtained from $h(r)$ by eliminating all the arguments that are free (i.e., denoted by an f in γ); then, $m.q'$ is the magic goal added to r.

The rewriting methods for supplementary magic predicates, and for the counting method, can be derived as simple modifications of the templates for magic sets. While the counting method is limited to the situation where we have only one recursive rule and this rule is linear, the other two methods are applicable whenever the binding passing property holds (see Exercise 9.7).

The magic sets method can also be used as the basis for detecting and handling the special cases of left-linear and right-linear rules. For instance, if we write the magic rules for Example 9.8, we obtain:

$$\begin{aligned} &\texttt{m.anc}(\texttt{tom}). \\ &\texttt{m.anc}(\texttt{Old}) \leftarrow \quad \texttt{m.anc}(\texttt{Old}). \end{aligned}$$

Obviously the recursive magic rule above is trivial and can be eliminated. Since the magic relation `anc` now contains only the value `tom`, rather than appending the magic predicate goal to the original rules, we can substitute this value directly into the rules. It is simple for a compiler to recognize the situation where the body and the head of the rule are identical, and then to eliminate the magic rule and perform the substitution.

Consider now the application of the magic sets method to Example 9.8. We obtain

$$\begin{aligned} &\texttt{m.anc}(\texttt{tom}). \\ &\texttt{m.anc}(\texttt{Mid}) \leftarrow && \texttt{parent}(\texttt{Old}, \texttt{Mid}), \texttt{m.anc}(\texttt{Old}). \\ &\texttt{anc}'(\texttt{Old}, \texttt{Young}) \leftarrow && \texttt{m.anc}(\texttt{Old}), \texttt{parent}(\texttt{Old}, \texttt{Young}). \\ &\texttt{anc}'(\texttt{Old}, \texttt{Young}) \leftarrow && \texttt{parent}(\texttt{Old}, \texttt{Mid}), \texttt{anc}'(\texttt{Mid}, \texttt{Young}), \\ & && \texttt{m.anc}'(\texttt{Old}). \\ &?\texttt{anc}'(\texttt{tom}, \texttt{Young}). \end{aligned}$$

Observe that the recursive rule defining `anc'` here plays no useful role. In fact, the second argument of `anc'` (i.e., `Young`) is simply copied from the goal to the head of the recursive rule. Moreover, once this second argument is dropped, then this rule simply revisits the magic set computation leading back to `tom`. Thus, every value of `Young` produced by the exit rule satisfies the query. Once the redundant recursive rule is eliminated, we obtain the following program:

```
m.anc(tom).
m.anc(Mid) ←          m.anc(Old), parent(Old, Mid).
anc'(Old, Young) ←    m.anc(Old), parent(Old, Young).
?anc'(tom, Young).
```

In general, for the recursive rule to be dropped, the following two conditions must hold: (1) all the recursive goals in the recursive rule have been used as chain goals (during the binding passing analysis), and (2) the free arguments in the recursive goal are identical to those of the head. These are simple syntactic tests for a compiler to perform. Therefore, the transformation between right-linear recursion and left-linear recursion can be compiled as a special subcase of the magic sets method.

9.6.4 Optimization Strategies and Execution Plan

Several variations are possible in the overall compilation and optimization strategy described in the previous sections. For instance, the requirement of having the unique binding passing property can be relaxed easily (see Exercise 9.9). The supplementary magic method can also be generalized to allow the passing of bindings between recursive goals in the same rule; however, the transformed programs so produced can be complex and inefficient to execute.

A topic that requires further research is query optimization. Most relational databases follow the approach of estimating the query cost under all possible join orders and then selecting the plan with the least-cost estimate. This approach is not commonly used in deductive database prototypes because of its prohibitive cost for large programs and the difficulty of obtaining reliable estimates for recursive predicates. Therefore, many systems use instead simple heuristics to select an order of execution for the goals. For instance, to select the next goal, precedence is given to goals that have more bound arguments and fewer unbound arguments than the other goals.

In other systems, the order of goal execution is that in which they appear in the rule (i.e., the Prolog convention also followed in the rule-goal graph of Figure 9.1). This approach leaves the control of execution in the hands of the programmer, with all the advantages and disadvantages that follow. A promising middle ground consists of using the optimization techniques of relational systems for simple rules and queries on base predicates, while letting the programmer control the execution of more complex programs, or predicates more remote from the base predicates.

Although different systems often use a different mix of recursive methods, they normally follow the same general approach to method selection. Basically, the different techniques, each with its specific applicability preconditions, are ranked in order of desirability; the first applicable method in the list is then selected. Therefore, the binding passing property is tested first, and if this is satisfied, methods such as those for left-linear and right-linear recursion are tried first; then if these fail, methods such as magic sets and supplementary magic are tried next. Several other techniques have been proposed for recursion, and novel approaches and refinements are being proposed frequently—although it is often difficult to evaluate the comparative effectiveness of the different techniques.

An additional generalization that should be mentioned allows some arguments of a goal to remain uninstantiated after its execution. In this approach, variables not bound by the execution of the goal will need to be bound by later goals, or will be returned to the head of the rule, and then to the calling goal, as unbound variables.

In addition to the global techniques discussed above, various optimizations of a local and specialized nature can be performed on Datalog-like languages. One such technique consists in avoiding the generation of multiple bindings for existential variables, such as variables that occur only once in a rule. Techniques for performing intelligent backtracking have also been used; these can, for example, simulate multiway joins in a tuple-at-a-time execution model. Therefore, many of the local optimization techniques used are specific to the low-level execution model adopted by the system; this, in turn, depends on many factors, including whether the system is primarily designed for data residing on secondary storage or data already loaded in main memory. These alternatives have produced the assortment of techniques and design choices explored by current deductive database prototypes.

9.7 Recursive Queries in SQL

The new SQL3 standards include support for recursive queries. For instance, the BoM program of Example 8.15 is expressible in SQL3, using the view construct as follows:

Example 9.20 Recursive views in SQL3

```
CREATE RECURSIVE view all_subparts(Major, Minor) AS
        SELECT PART SUBPART
        FROM assembly
    UNION
```

```
SELECT all.Major assb.SUBPART
FROM all_subparts all, assembly assb
WHERE all.Minor= assb.PART
```

The **SELECT** statement before **UNION** is obviously equivalent to the exit rule in Example 8.15, while the **SELECT** statement after **UNION** corresponds to the recursive rule. Therefore we will refer to them as *exit select* and *recursive select*, respectively.

Since all_subparts is a virtual view, an actual query on this view is needed to materialize the recursive relation or portions thereof. For instance, the query of Example 9.21 requests the materialization of the whole relation.

Example 9.21 Materialization of the view of Example 9.20

```
SELECT *
FROM all_subparts
```

The **WITH** construct provides another way, and a more direct one, to express recursion in SQL3. For instance, a query to find all the superparts using 'top_tube' can be expressed as follows:

Example 9.22 Find the parts using 'top_tube'

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

9.7.1 Implementation of Recursive SQL Queries

The compilation techniques developed for Datalog apply directly to recursive SQL queries. For instance, the query of Example 9.21 on the view defined in Example 9.20 requires the materialization of the whole transitive closure, and can thus be implemented efficiently using the differential fixpoint Algorithm 9.5. Then, $T_E(S')$ and $T_R(S')$ are, respectively, computed from the exit select and the recursive select in Example 9.20. Here too, the computation of $T_R(S') - S'$ will be improved using the differential fixpoint technique. In

fact, this step is simple to perform since there is only one recursive relation in the **FROM** clause of Example 9.20; therefore, this is a case of linear recursion. Thus, the recursive relation all_subparts in the **FROM** clause is replaced with δall_subparts, which contains new tuples generated in the previous iteration of Algorithm 9.5.

Consider now Example 9.22. This requires the passing of the condition Minor= 'top_tube' into the recursive SQL query defined using **WITH**. Now, the recursive select in Example 9.22 uses right-linear recursion, whereby the second argument of the recursive relation is copied unchanged by T_R. Thus, the condition Minor = 'top_tube' can simply be attached unchanged to the **WHERE** clause of the exit select and the recursive select, yielding the following equivalent SQL program:

Example 9.23 Specialization of the query of Example 9.22

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

However, say that the same query is expressed against the virtual view of Example 9.20, as follows:

```
SELECT *
FROM all_subparts
WHERE Minor = 'top_tube'
```

Since all_subparts is defined in Example 9.20 using left-linear recursion, the addition of the condition Minor = 'top_tube' to the recursive select would not produce an equivalent query. Instead, the SQL compiler must transform the original recursive select into its right-linear equivalent before the condition Minor = 'top_tube' can be attached to the **WHERE** clause. The compilation techniques usable for such transformations are basically those previously described for Datalog.

9.8 Bibliographic Notes

The notion of differential fixpoint based on the "derivatives" of set-valued operations was proposed in different contexts by [172, 330]. The use of the technique for improving naive fixpoint computation was studied by Bayer [47] and by Bancilhon, who coined the name "semi-naive" [28]. The idea of direct differentiation on the rules was introduced by Balbin [24], and the rewriting presented in this chapter is due to Saccà and Zaniolo [373].

The best-known algorithm for unification is due to Martelli and Montanari [277].

The magic sets technique was introduced by Bancilhon, Maier, Sagiv, and Ullman [29]; supplementary predicates were introduced by Saccà and Zaniolo [373]. The generalized magic sets method is credited to Beeri and Ramakrishnan [49]. The counting algorithm was proposed in [29] and generalized in [374]. Approaches to ensure that the counting method can deal with cycles are presented in [372, 203, 16, 194]. An evaluation of the counting method and other techniques is presented in [275].

Techniques for efficient support of recursion with a combination of top-down, bottom-up, and memoing have been the focus of a large body of research. Here we only mention the work by Henschen and Naqvi [212], the query-subquery approach by Vielle [449], and the approach based on the parsing technique called "Early Deduction" by Pereira and Warren [336]. A survey of early techniques was presented in [30].

The notion of "separable" recursion introduced by Naughton [303] includes left-linear rules, right-linear rules, and mixed left- and right-linear rules, which can be implemented by a single fixpoint. Similar recursive rules are also treated by Han [207], using the notion of chain queries. The simplified compilation presented here for left-linear and right-linear rules are based on the techniques presented in [194].

The NAIL! project introduced the use of rule-goal graph for compilation and the use of capture rules for optimization [296, 295]. Various optimization techniques for Datalog were also explored in [104, 348]. Techniques for safety analysis of bottom-up and top-down queries are discussed in [464, 347]. A more general notion of safety is discussed in [445].

Among the more influential prototypes of deductive databases, we can mention NAIL!/Glue [296, 338], LDL/LDL++ [105, 469], and CORAL [349, 350]. A more extensive overview of different systems can be found in [351].

Many novel applications of deductive databases have been discussed in the literature. An early overview was presented by Tsur [436], and a more recent book on the subject is [346].

The recursive extensions of SQL3, discussed here, are based on the proposal by Finkelstein, Mattos, Mumick, and Pirahesh [168].

9.9 Exercises

9.1. Use the `emp_all_mgrs(Eno, AllManagers)` relation of Exercise 8.12 to construct a combined list of managers for employees '`Joe Doe`' and '`Tom Jones`'. First write the rules to perform a bottom-up list append of the manager list for the first employee with that of second employee. Then refine this program to ensure that no duplicate manager is inserted in the list.

9.2. A better-known list append is the top-down one used in Prolog. Write (top-down) recursive rules to compute `app(L1, L2, L3)`, which appends two lists `L1` and `L2`, giving `L3`.

9.3. Write Prolog and Datalog rules to generate all positive integers up to a given integer K—in ascending and descending order.

9.4. Write nonlinear Prolog rules to compute the first K Fibonacci numbers in a bottom-up fashion.

9.5. Write linear rules to compute efficiently the first K Fibonacci numbers in Prolog. Solve the same problem for Datalog.

9.6. Describe how the binding passing analysis will be performed on Exercise 9.2, and which compilation methods can be used. Transform the original programs according to compilation methods selected.

9.7. An arithmetic expression such as $y \times y + c$ might be parsed and represented internally as a complex term such as

$$\texttt{plus}(\texttt{times}(\texttt{var}(\texttt{y}), \texttt{var}(\texttt{y})), \texttt{cost}(\texttt{c})).$$

Write rules to perform a symbolic differentiation of such expressions. Thus our original query goal might be ?`derivation($Expr, Derivt)`. If the previous expression is given as the first argument, the second argument returned should be

`plus(plus(times(cost(1), var(y)), times(var(y), cost(1)), cost(0))).`

Just give the basic top-down rules for addition and multiplication. The most direct formulation requires nonlinear rules that cannot be supported in Prolog; explain why.

9.8. Perform the binding analysis for Exercise 9.7, and show how this leads to the application of the magic sets method or supplementary magic technique. Rewrite the original programs according to both methods.

9.9. Explain how the magic sets and related methods are also applicable to programs that have the binding passing property, but not the unique binding passing property. Apply the magic sets method to the following example:

```
?sg(marc, Who).
sg(X, Y) ←        parent(XP, X), sg(YP, XP), parent(YP, Y).
sg(A, A).
```

Chapter 10

Database Updates and Nonmonotonic Reasoning

Two important problems have been left open in previous chapters. One is how to relax the restriction that negation and other nonmonotonic constructs can only appear in stratified programs, which represents a serious limitation in many practical applications. The other problem is that, so far, we have neglected the dynamic aspects of database systems, such as database updates and active database rules. In this chapter, we fill these lacunas and develop a unified approach to reasoning with nonmonotonic knowledge, nondeterministic constructs, database updates, active rules, and database histories.

10.1 Nonmonotonic Reasoning

The issue of nonmonotonic reasoning is a difficult problem that databases share with several other areas, including artificial intelligence, knowledge representation, and logic programming. Much previous research has focused on the problems of correct semantics and logical consistency. To address the efficiency and scalability requirements of databases, we will also stress issues such as computational complexity, expressive power, amenability to efficient implementation, and usability.

Negation represents the quintessential nonmonotonic construct; as we shall see later, issues regarding other nonmonotonic constructs, such as set aggregates or updates, can be reduced to issues regarding negation.

Most of the technical challenges posed by negation follow from the fact that information systems normally do not store negative information ex-

plicitly. Rather, they normally store only positive information, and negative information is derived implicitly via some form of default, such as the closed-world assumption. For instance, in our university database of Example 8.1, we see that only courses actually taken by students are stored. Thus, if for a certain student there is no "cs123" entry in the database, then the conclusion follows that the student has not taken the course.

Two different views of the world are possible.

- *Open world:* What is not part of the database or the program is assumed to be unknown.
- *Closed world:* What is not part of the database or the program is assumed to be false.

Databases and other information systems adopt the closed-world assumption (CWA). Thus, if $\mathtt{p}$ is a base predicate with n arguments, then the CWA prescribes that $\neg\mathtt{p}(\mathtt{a}_1, \ldots, \mathtt{a}_n)$ holds iff $\mathtt{p}(\mathtt{a}_1, \ldots, \mathtt{a}_n)$ is not true, that is, is not in the database. This assumption is natural and consistent, provided that a *unique name axiom* is also satisfied. The unique name assumption specifies that two constants in the database cannot stand for the same semantic object. According to the unique name assumption, for instance, the absence of `coolguy('Clark Kent')` from the database implies `¬coolguy('Clark Kent')`, even though the database contains a fact `coolguy('Superman')`. This negative conclusion does not follow in a system that does not adopt the unique name assumption, and, for instance, treats `'Superman'` and `'Clark Kent'` as different references to the same underlying entity.[1]

As we shall see next, the extension of the CWA to programs where rules contain negation might lead to inconsistencies. The CWA for a positive program P is as follows:

Definition 10.1 *Let P be a positive program; then for each atom $a \in B_P$,*
(i) a is true iff $a \in T_P^{\uparrow\omega}(\emptyset)$, and
(ii) $\neg a$ is true iff $a \notin T_P^{\uparrow\omega}(\emptyset)$.

Definition 10.1 basically states that every conclusion derivable from the given program is true, and everything else is false. This definition is consistent when P is a positive program. However, contradictions may arise when P is a general program where rules are allowed to contain negated goals.

[1] Lois Lane's unique name assumption has provided the theme and romantic plot for many Superman movies. Also Herbrand interpretations follow this approach by letting constants represent themselves.

Consider for instance the classical paradox of the village where the barber shaves everyone who does not shave himself:

Example 10.1 Every villager who does not shave himself is shaved by the barber

```
shaves(barber, X) ← villager(X), ¬shaves(X, X).
shaves(miller, miller).
villager(miller).
villager(smith).
villager(barber).
```

There is no problem with `villager(miller)`, who shaves himself, and therefore does not satisfy the body of the first rule. For `villager(smith)`, given that `shaves(smith, smith)` is not in our program, we can assume that `¬shaves(smith, smith)`. Then, the body of the rule is satisfied, and the conclusion `shaves(barber, smith)` is reached. There is no problem with this conclusion, since it is consistent with all the negative assumptions previously made.

However, consider `villager(barber)`. If we make the assumption that `¬shaves(barber, barber)`, the rule yields `shaves(barber, barber)`, which directly contradicts the initial assumption. However, if we do not assume `¬shaves(barber, barber)`, then we cannot derive `shaves(barber, barber)` using this program. Therefore, by the CWA policy we must conclude that `¬shaves(barber, barber)`, which is again a contradiction. There is no way out of this paradox, and thus, there is no reasonable semantics for self-contradictory programs, such as that of Example 10.1, under the CWA. The *stable model semantics* discussed next characterizes programs that are free of such contradictions.

Definition 10.2 *Stability transformation. Let P be a program and $I \subseteq B_P$ be an interpretation of P. Then $ground_M(P)$ denotes the program obtained from $ground(P)$ by the following transformation, called the* stability transformation*:*

1. *Remove every rule having as a goal some literal $\neg q$ with $q \in I$.*
2. *Remove all negated goals from the remaining rules.*

Example 10.2 A program $P = ground(P)$

$$p \leftarrow \neg q$$

$$q \leftarrow \neg p$$

For the program in Example 10.2, the stability transformation yields the following:

For $I = \emptyset$, $ground_I(P)$ is $p.\ \ q.$

For $I = \{p\}$, $ground_I(P)$ is $p.$

For $I = \{p, q\}$, $ground_I(P)$ is the empty program.

Definition 10.3 *Stable models. Let P be a program with model M. M is said to be a* stable model *for P when M is the least model of $ground_M(P)$.*

Observe that $ground_M(P)$ is a positive program, by construction. So, it has a least model that is equal to $T^{\uparrow\omega}(\emptyset)$, where T here denotes the immediate consequence operator of $ground_M(P)$.

Every stable model for P is a minimal model for P and a minimal fixpoint for T_P; however, minimal models or minimal fixpoints need not be stable models, as in the example that follows:

Example 10.3 **$M = \{a\}$ is the only model and fixpoint for this program**

$$r_1 : \mathtt{a} \leftarrow \ \neg\mathtt{a}.$$
$$r_2 : \mathtt{a} \leftarrow \ \mathtt{a}.$$

M is the only model and fixpoint for the program in Example 10.3. But $ground_M(P)$ contains only rule r_2; thus, its least model is the empty set, and M is not a stable model.

A given program can have no stable model, a unique stable model, or multiple stable models. For instance, the program in Example 10.1 has no stable model, but it has a unique stable model after we eliminate the fact `villager(barber)`. The program of Example 10.3 has no stable model. The program of Example 10.2 has two stable models: $M_1 = \{p\}$ and $M_2 = \{q\}$. Thus there are two symmetric ways to give a logical meaning to this program: one where p is true and q is false, and the other where p is false and q is true. Since either solution can be accepted as the meaning of the program, we see that stable model semantics also brings the notion of nondeterminism to logic-based languages. This topic will be revisited in later sections.

The notion of stable models can be defined directly using a modified version of the immediate consequence operator (ICO) as follows: With r being a rule of P, let $h(r)$ denote the head of r, $gp(r)$ denote the set of positive goals of r, and $gn(r)$ denote the set of negated goals of r without their negation sign. For instance, if $r : a \leftarrow b, \neg c, \neg d.$, then $h(r) = a$, $gp(r) = \{b\}$, and $gn(r) = \{c, d\}$.

Definition 10.4 *Let P be a program and $I \subseteq B_P$. Then the* explicit negation ICO *for P under a set of negative assumptions $N \subseteq B_P$ is defined as follows:*

$$\Gamma_{P(N)}(I) = \{h(r) \mid r \in ground(P),\ gp(r) \subseteq I,\ gn(r) \subseteq N\}$$

We will keep the notation T_P to denote the *implicit negation ICO* of P, defined as follows:

$$T_P(I) = \Gamma_{P(\overline{I})}(I),\ \text{where } \overline{I} = B_P - I$$

While Γ can also be viewed as a two-place function (on I and N), in the following theorem we view it as a function of I only, since N is kept constant (the proof of the theorem follows directly from the definitions):

Theorem 10.5 *Let P be a logic program with Herbrand base B_P and $\overline{M} = B_P - M$. Then, M is a stable model for P iff*

$$\Gamma^{\uparrow\omega}_{P(\overline{M})}(\emptyset) = M$$

Thus, Theorem 10.5 states that M is a stable model if it can be obtained as the ω-power of the explicit negation ICO, where the set of false atoms is kept constant and equal to the set of atoms not in M. This theorem can be used to check whether an interpretation I is a stable model without having first to construct $ground_P(I)$. Furthermore, the computation of the ω-power of the positive consequence operator has polynomial data complexity (see Exercise 10.1); thus, checking whether a given model is stable can be done in polynomial time. However, deciding whether a given program has a stable model is, in general, $\mathcal{NP}$-complete; thus, finding any such model is $\mathcal{NP}$-hard.

Therefore, much research work has been devoted to the issue of finding polynomial-time algorithms that compute stable models for classes of programs of practical interest. The challenges posed by this problem can be appreciated by observing how techniques, such as SLD-resolution, that work well for positive programs run into difficulties when faced with programs with negation.

Prolog and other top-down systems use SLD-resolution with negation by failure (SLD-NF) for negative programs. In a nutshell, SLD-NF operates as follows: When presented with a goal $\neg g$, SLD-NF tries instead to prove g. Then, if g evaluates to true, $\neg g$ evaluates to false; however, if g evaluates to false, $\neg g$ evaluates to true. The answers returned by the SLD-resolution are correct in both these cases, but there is also the case where SLD-resolution

flounders in an infinite loop, and then no answer is returned. Unfortunately, this case is common when the Herbrand universe is infinite and can also occur when the universe is finite. For instance, if the program consists of only one rule $p \leftarrow p$, then the query $?\neg p$ flounders in an infinite loop and SLD-NF never returns any answer. Since the least model of the program $p \leftarrow p$ is the empty set, $?\neg p$ should evaluate to true under the CWA. Therefore, this example illustrates how SLD-NF does not provide a complete implementation of the CWA even for simple positive programs. A similar problem occurs when a goal like $\neg$`anc(marc, mary)` is expressed against the program of Example 9.5. SLD-NF flounders if `marc` and `mary` are nodes in a directed cycle of a graph corresponding to the `parent` relation. Both these two queries, where SLD-NF fails, can be supported by the iterated fixpoint used in the bottom-up computation of stratified programs.

10.2 Stratification and Well-Founded Models

Stable model semantics is very powerful, and various nonmonotonic knowledge representation formalisms such as prioritized circumscription, default theory, and autoepistemic logic can be reduced to it.

But given the exponential complexity of computing a stable model, current research is seeking more restrictive classes of programs capable of expressing the intended applications while having stable models computable in polynomial time. In this chapter, we consider stratified and locally stratified programs, and the notion of well-founded models.

A stratified program has a stable model that can be computed using the iterated fixpoint computation (Algorithm 9.2). Furthermore, we will prove later that such a model is unique.

Theorem 10.6 *Let P be a stratified program. Then P has a stable model that is equal to the result of the iterated fixpoint procedure.*

Proof. Let Σ be a stratification for P, and let M be the result of the iterated fixpoint on P according to Σ. Since the iterated fixpoint on $ground(P)$ according to Σ also yields M, let $r \in ground(P)$ be a rule used in the latter computation, where $h(r)$ belongs to a stratum i. If $\neg g$ is a goal of r, then the predicate name of g belongs to a stratum lower than i, and thus g cannot be generated by the iterated fixpoint computation of strata $\geq i$. Therefore, for each rule r used in the iterated fixpoint computation of $ground(P)$, $r' \in ground_M(P)$, where r' is the rule obtained from r by removing its negated goals. Therefore, the iterated fixpoint computation on the iterated

fixpoint on $ground_M(P)$ according to Σ also yields M. Therefore, M is a stable model. □

Since the class of stratified programs is too restrictive in many applications, we now turn to the problem of going beyond stratification and allowing the usage of negated goals that are mutually recursive with the head predicates.

10.2.1 Locally Stratified Programs

The notion of local stratification provides a generalization where atoms are stratified on the basis of their argument values, in addition to the names of their predicates.

Definition 10.7 *Local stratification. A program P is locally stratifiable iff B_P can be partitioned into a (possibly infinite) set of strata $S_0, S_1, \ldots$, such that the following property holds: For each rule r in $ground(P)$ and each atom g in the body of r, if $h(r)$ and g are, respectively, in strata S_i and S_j, then*

(i) $i \geq j$ if $g \in pg(r)$, and

(ii) $i > j$ if $g \in ng(r)$.

Example 10.4 A locally stratified program defining integers

```
even(0).
even(s(J)) ← ¬even(J).
```

The program in Example 10.4 has an obvious local stratification, obtained by assigning `even(0)` to S_0, `even(s(0))` to S_1, and so on.

The program in Example 10.5 attempts an alternative definition of integers; this program is not locally stratified (see Exercise 10.6).

Example 10.5 A program that is not locally stratified

```
even(0).
even(J) ← ¬even(s(J)).
```

Theorem 10.8 *Every locally stratified program has a stable model that is equal to the result of the iterated fixpoint computation.*

The proof of this last theorem is the same as the proof of Theorem 10.6. This underscores the conceptual affinity existing between stratified programs and locally stratified programs. The two classes of programs, however, behave very differently when it comes to actual implementation. Indeed, a program P normally contains a relatively small number of predicate names. Thus, the verification that there is no strong component with negated arcs in $pdg(P)$ and the determination of the strata needed for the iterated fixpoint computation are easy to perform at compile-time. However, the question of whether a given program can be locally stratified is undecidable when the Herbrand base of the program is infinite. Even when the universe is finite, the existence of a stable model cannot be checked at compile-time, since it often depends on the database content. For instance, in Example 10.1, the existence of a stable model depends on whether `villager(barber)` is in the database.

Much research work has been devoted to finding general approaches for the efficient computation of nonstratified programs. The concept of well-founded models represents a milestone in this effort.

10.2.2 Well-Founded Models

The basic equality $lfp(T_P) = T_P^{\uparrow\omega}$, which was the linchpin of the bottom-up computation, no longer holds in the presence of negation. One possible solution to this dilemma is to derive from T_P a new operator that is monotonic. This leads to the notion of alternating fixpoint and well-founded models, discussed next.

The operator $\Gamma_{P(N)}^{\uparrow\omega}(\emptyset)$ is monotonic in N. Thus

$$S_P(N) = B_P - \Gamma_{P(N)}^{\uparrow\omega}(\emptyset)$$

is antimonotonic in N (i.e., $S_P(N') \subseteq S_P(N)$ for $N' \supseteq N$). Therefore, the composition of an even number of applications of S_P yields a monotonic mapping, while the composition of an odd number of applications yields an antimonotonic mapping. In particular,

$$A_P(N) = S_P(S_P(N))$$

is monotonic in N. Thus, by Knaster-Tarski's theorem, A_P has a least fixpoint $lfp(A_P)$. Actually, A_P might have several fixpoints:

Lemma 10.9 *Let $(M, \overline{M})$ be a dichotomy of B_P. Then, M is a stable model for P iff $\overline{M}$ is a fixpoint for S_P. Also, every stable model for P is a fixpoint of A_P.*

Proof. By Theorem 10.5, M is a stable model for P iff $\Gamma_{P(\overline{M})}^{\uparrow\omega}(\emptyset) = M$. This equality holds iff $B_P - \Gamma_{P(\overline{M})}^{\uparrow\omega}(\emptyset) = B_P - M = \overline{M}$, i.e., iff $S_P(\overline{M}) = \overline{M}$. Finally, every fixpoint for S_P is also a fixpoint for A_P. □

The least fixpoint $lfp(A_P)$ can be computed by (possibly transfinite) applications of A_P. Furthermore, every application of A_P in fact consists of two applications of S_P. Now, since $A_P^{\uparrow n-1}(\emptyset) \subseteq A_P^{\uparrow n}(\emptyset)$, the even powers of S_P

$$A_P^{\uparrow n}(\emptyset) = S_P^{\uparrow 2\times n}(\emptyset)$$

define an ascending chain. The odd powers of S_P

$$S_P(A_P^{\uparrow n}(\emptyset)) = S_P^{\uparrow 2\times n+1}(\emptyset)$$

define a descending chain. Furthermore, it is easy to show that every element of the descending chain is $\supseteq$ than every element of the ascending chain. Thus we have an increasing chain of underestimates dominated by a decreasing chain of overestimates. If the two chains ever meet,[2] they define the (total) well-founded model[3] for P.

Definition 10.10 *Well-founded model. Let P be a program and W be the least fixpoint for A_P. If $S_P(W) = W$, then $B_P - W$ is called the* well-founded model *for P.*

$B_P - S_P(\overline{M}) = B_P - \overline{M} = M$. But $B_P - S_P(\overline{M}) = \Gamma_{P(\overline{M})}^{\uparrow\omega}(\emptyset)$.

Theorem 10.11 *Let P be a program with well-founded model M. Then M is a stable model for P, and P has no other stable model.*

Proof. The fact that M is a stable model was proven in Lemma 10.9. Now, by the same lemma, if N is another stable model, then $\overline{N}$ is also a fixpoint for A_P; in fact, $\overline{N} \supseteq \overline{M}$, since $\overline{M}$ is the least fixpoint of A_P. Thus, $N \subseteq M$; but, $N \subset M$ cannot hold, since M is a stable model, and every stable model is a minimal model. Thus $N = M$. □

The computation of $A_P^{\uparrow\omega}(\emptyset)$ is called the *alternating fixpoint computation.* The alternating fixpoint computation for the program of Example 10.2 is shown in Example 10.6.

[2] Including the situation where they meet beyond the first infinite ordinal.

[3] The term *well-founded model* is often used to denote the *partial well-founded model*, defined as having as negated atoms $M^- = lfp(A_P)$ and positive atoms $M^+ = B_P - S_P(M^-)$; thus, the atoms in $B_P - (M^+ \cup M^-)$ are undefined in the partial well-founded model, while this set is empty in the total well-founded model.

Example 10.6 The alternating fixpoint computation for Example 10.2

$$S_P(\emptyset) = \{p, q\}$$

$$A_P(\emptyset) = S_P(S_P(\emptyset)) = S_P(\{p, q\}) = \emptyset$$

Then, the situation repeats itself, yielding

$$A_P^{\uparrow k}(\emptyset) = A_P(\emptyset) \subset S_P(A_P^{\uparrow k}(\emptyset))$$

Since the overestimate and underestimate never converge, this program does not have a (total) well-founded model.

Indeed, Example 10.2 has two stable models; thus, by Theorem 10.11, it cannot have a well-founded model.

Stratified and locally stratified programs always have a well-founded model (and therefore a unique stable model) that can be computed using the alternating fixpoint procedure:

Theorem 10.12 *Let P be a program that is stratified or locally stratified. Then P has a well-founded model.*

Proof (Sketch). If P is a program that is stratified or locally stratified, then the alternating fixpoint procedure emulates the stratified fixpoint procedure. □

While the notion of a well-founded model is significant from a conceptual viewpoint, it does not provide a simple syntactic criterion that the programmer can follow (and the compiler can exploit) when using negation in recursive rules, so as to ensure that the final program has a clear semantics. The objective of achieving local stratification through the syntactic structure of the rules can be obtained using Datalog_{1S}.

10.3 Datalog_{1S} and Temporal Reasoning

In Datalog_{1S}, predicates have a distinguished argument, called the *temporal argument*, where values are assumed to be taken from a discrete temporal domain. The discrete temporal domain consists of terms built using the constant 0 and the unary function symbol +1 (written in postfix notation). For the sake of simplicity, we will write n for

$$(\ldots((0\overbrace{+1)+1)\ldots+1)}^{n \text{ times}}$$

Also, if T is a variable in the temporal domain, then T, $T+1$, and $T+n$ are valid temporal terms, where $T+n$ is again a shorthand for

$$(\ldots((T\overbrace{+1)+1)\ldots+1)}^{n\ times}$$

The following Datalog program models the succession of seasons:

Example 10.7 The endless succession of seasons

```
quarter(0, winter).
quarter(T + 1, spring) ← quarter(T, winter).
quarter(T + 1, summer) ← quarter(T, spring).
quarter(T + 1, fall) ← quarter(T, summer).
quarter(T + 1, winter) ← quarter(T, fall).
```

Therefore, Datalog$_{1S}$ provides a natural formalism for modeling events and history that occur in a discrete time domain (i.e., a domain isomorphic to integers). The granularity of time used, however, depends on the application. In the previous example, the basic time granule was a season. In the next example, which lists the daily schedule of trains to Newcastle, time granules are hours.

Trains for Newcastle leave every two hours, starting at 8:00 AM and ending at 10:00 PM (last train of the day). Here we use midnight as our initial time, and use hours as our time granules. Then we have the following daily schedule:

Example 10.8 Trains for Newcastle leave daily at 800 hours and then every two hours until 2200 hours (military time)

```
before22(22).
before22(H) ← before22(H + 1).
leaves(8, newcastle).
leaves(T + 2, newcastle) ← leaves(T, newcastle),
                           before22(T).
```

Thus the query ?`leaves(When, newcastle)` will generate the daily departure schedule for Newcastle.

This example also illustrates how Datalog$_{1S}$ models the notion of before 10 PM (2200 hours in military time), and after 8 AM.

Datalog_{1S} is standard Datalog, to which a particular temporal interpretation is attached. In fact, we have already encountered Datalog_{1S} programs. For instance, the programs in Examples 10.4 and 10.5 are Datalog_{1S} programs where $s(J)$ is used instead of $J+1$ to model the notion of successor (actually the name Datalog_{1S} originated from this alternate notation). Also the programs in Example 9.15 are Datalog_{1S}, since the pairs $J-1$ and J in the counting rules can be replaced by I and $I+1$ without changing their meaning.

Remarkably, Datalog_{1S} represents as powerful a language for temporal reasoning as special-purpose temporal languages with modal operators. Take for instance Propositional Linear Temporal Logic (PLTL).

PLTL is based on the notion that there is a succession of states $H = (S_0, S_1, \ldots)$, called a *history*. Then, modal operators are used to define in which states a predicate p holds true.

For instance, the previous example of trains to Newcastle can be modeled by a predicate *newcstl* that holds true in the following states: S_8, S_{10}, $S_{12}, S_{14}, S_{16}, S_{18}, S_{20}, S_{22}$, and it is false everywhere else.

Then temporal predicates that hold in H are defined as follows:

1. *Atoms:* Let p be an atomic propositional predicate. Then p is said to hold in history H when p holds in H's initial state S_0.

In addition to the usual propositional operators $\vee, \wedge$, and $\neg$, PLTL offers the following operators:

2. *Next:* Next p, denoted $\bigcirc p$, is true in history H, when p holds in history $H_1 = (S_1, S_2, \ldots)$.

   Therefore, $\bigcirc^n p$, $n \geq 0$, denotes that p is true in history $(S_n, S_{n+1}, \ldots)$.

3. *Eventually:* Eventually q, denoted $\mathcal{F}q$, holds when, for some n, $\bigcirc^n q$.

4. *Until:* p until q, denoted $p\ \mathcal{U}\ q$, holds if, for some n, $\bigcirc^n q$, and for every state $k < n$, $\bigcirc^k p$.

Other important operators can be derived from these. For instance, the fact that q will never be true can simply be defined as $\neg\mathcal{F}q$; the fact that q is always true is simply described as $\neg\mathcal{F}(\neg q)$); the notation $\mathcal{G}q$ is often used to denote that q is always true.

The operator p before q, denoted $p\mathcal{B}q$ can be defined as $\neg((\neg p)\ \mathcal{U}\ q)$—that is, it is not true that p is false until q.

PLTL finds many applications, including temporal queries and proving properties of dynamic systems. For instance, the question "Is there a train

to Newcastle that is followed by another one hour later?" can be expressed by the following query:

$$?\mathcal{F}(newcstl \wedge \bigcirc newcstl)$$

Every query expressed in PLTL can also be expressed in propositional Datalog_{1S} (i.e., Datalog with only the temporal argument). For instance, the previous query can be turned into the query ?`pair_to_newcstl` where

$$\texttt{pair\_to\_newcstl} \leftarrow \texttt{newcstl(J)} \wedge \texttt{newcstl(J + 1)}.$$

Therefore, the interpretation "for some J" assigned to the temporal argument J of a Datalog_{1S} predicate is sufficient to model the operator $\mathcal{F}$ of PLTL, while $\bigcirc$ is now emulated by $+1$.

The translation of the other operators, however, is more complex. Say, for instance, that we want to model $p\ \mathcal{U}\ q$. By the definition, p must be true at each instant in history, until the first state in which q is true.

Example 10.9 $p\ \mathcal{U}\ q$ in Datalog_{1S}

$$\begin{array}{ll}
\texttt{post_q(J + 1)} \leftarrow & \texttt{q(J)}. \\
\texttt{post_q(J + 1)} \leftarrow & \texttt{post_q(J)}. \\
\texttt{first_q(J)} \leftarrow & \texttt{q(J)}, \neg\texttt{post_q(J)}. \\
\texttt{pre_first_q(J)} \leftarrow & \texttt{first_q(J + 1)}. \\
\texttt{pre_first_q(J)} \leftarrow & \texttt{pre_first_q(J + 1)}. \\
\texttt{fail_p_Until_q} \leftarrow & \texttt{pre_first_q(J)}, \neg\texttt{p(J)}. \\
\texttt{p_Until_q} \leftarrow & \texttt{pre_q(0)}, \ \neg\texttt{fail_p_Until_q}.
\end{array}$$

Therefore, we used recursion to reason back in time and identify all states in history that precede the first occurrence of `q`. Then, `p` $\mathcal{U}$ `q` was defined using double negation, yielding a program with stratified negation (although this is not necessary; see Exercise 10.8). A similar approach can be used to express other operators of temporal logic. For instance, `p` $\mathcal{B}$ `q` can be defined using the predicates in Example 10.9 and the rule

$$\texttt{p\_Before\_q} \leftarrow \texttt{p(J)}, \texttt{pre\_first\_q(J)}.$$

10.4 XY-Stratification

The main practical limitation of semantics based on concepts such as well-founded models is that there is no simple way to decide whether a program

obeys such a semantics, short of executing the program. This is in sharp contrast with the concept of stratified negation, where a predicate dependency graph free of cycles provides a simple criterion for the programmer to follow in writing the program, and for the compiler to check and use in validating and optimizing the execution of the program. We will next discuss a particular mixture of Datalog_{1S} and stratification that has great expressive power but preserves much of the simplicity of stratified programs.

For instance, the ancestors of `marc`, and the number of generations that separates them from `marc`, can be computed using the following program, which also includes the differential fixpoint improvement:

Example 10.10 Ancestors of marc and the generation gap including the differential fixpoint improvement

r_1 : delta_anc(0, marc).
r_2 : delta_anc(J + 1, Y) ← delta_anc(J, X), parent(Y, X), ¬all_anc(J, Y).
r_3 : all_anc(J + 1, X) ← all_anc(J, X).
r_4 : all_anc(J, X) ← delta_anc(J, X).

This program is locally stratified by the first argument in `anc` that serves as temporal argument. The zeroth stratum consists of atoms of nonrecursive predicates such as `parent` and of atoms that unify with `all_anc(0, X)` or `delta_anc(0, X)`, where `X` can be any constant in the universe. The k^{th} stratum consists of atoms of the form `all_anc(k, X)`, `delta_anc(k, X)`. Thus, this program is locally stratified, since the heads of recursive rules belong to strata that are one above those of their goals. Also observe that the program of Example 10.4 has this property, while the program of Example 10.5 does not.

So far, we have studied the case where all recursive atoms with the same temporal argument belong to the same stratum. This structure can be generalized by partitioning atoms with the same temporal argument into multiple substrata. For instance, in Example 10.10, a strict stratification would place `delta_anc(k, X)` in a stratum lower than `all_anc(k, X)`. In fact, if the goal ¬`delta_anc(J + 1, X)` is added to the third rule (this will not change the meaning of the program), having the former atoms in a lower stratum becomes necessary to preserve local stratification.

From these examples, therefore, we can now describe the syntactic structure of our programs as follows:

Definition 10.13 *XY-programs. Let P be a set of rules defining mutually recursive predicates. Then we say that P is an XY-program if it satisfies the following conditions:*

1. *Every recursive predicate of P has a distinguished temporal argument.*
2. *Every recursive rule r is either an X-rule or a Y-rule, where*
   - *r is an X-rule when the temporal argument in every recursive predicate in r is the same variable (e.g., J),*
   - *r is a Y-rule when (i) the head of r has as temporal argument $J+1$, where J denotes any variable, (ii) some goal of r has as temporal argument J, and (iii) the remaining recursive goals have either J or $J+1$ as their temporal arguments.*

For instance, the program in Example 10.10 is an XY-program where r_4 is an X-rule while r_2 and r_3 are Y-rules.

Therefore, exit rules establish initial values for the temporal argument; then the X-rules are used for reasoning within the same state (i.e., the same value of temporal argument) while the Y-rules are used for reasoning from one state to the successor state.

There is a simple test to decide whether an XY-program P is locally stratified. The test begins by labeling the recursive predicates in P to yield the *bi-state program* P_{bis}, computed as follows: For each $r \in P$,

1. Rename all the recursive predicates in `r` that have the same temporal argument as the head of r with the distinguished prefix `new_` .
2. Rename all other occurrences of recursive predicates in `r` with the distinguished prefix `old_`.
3. Drop the temporal arguments from the recursive predicates.

For instance, the bi-state version for the program in Example 10.10 is as follows:

Example 10.11 The bi-state version of the program in Example 10.10

```
new_delta_anc(marc).
new_delta_anc(Y) ←    old_delta_anc(X), parent(Y, X),
                      ¬old_all_anc(Y).
new_all_anc(X) ←      new_delta_anc(X).
new_all_anc(X) ←      old_all_anc(X).
```

Definition 10.14 *Let P be an XY-program. P is said to be XY-stratified when P_{bis} is a stratified program.*

The program of Example 10.11 is stratified with the following strata: $\mathtt{S}_0 = \{\mathtt{parent}, \mathtt{old\_all\_anc}, \mathtt{old\_delta\_anc}\}$, $\mathtt{S}_1 = \{\mathtt{new\_delta\_anc}\}$, and $\mathtt{S}_2 = \{\mathtt{new\_all\_anc}\}$. Thus, the program in Example 10.10 is locally stratified.

Theorem 10.15 *Let P be an XY-stratified program. Then P is locally stratified.*

Proof. Let Σ be a stratification of P_{bis} in $n+1$ strata numbered from 0 to n, where we can assume, without loss of generality, that if $\mathtt{p}$ is a recursive predicate, then $\mathtt{old\_p}$ along with every nonrecursive predicate belongs to stratum 0. Then, a local stratification for P can be constructed by assigning every recursive atom with predicate name, say, $\mathtt{q}$, to the stratum $j \times n + k$, where k is the stratum of Σ to which $\mathtt{new\_q}$ belongs and j is the temporal argument in $\mathtt{q}$. Nonrecursive predicates are assigned to stratum 0. Then, by construction, the head of every rule $r \in \mathit{ground}(P)$ belongs to a stratum that is higher than the strata containing positive goals of r and strictly higher than the strata containing negated goals of r. □

Thus, the program of Example 10.10 is locally stratified with strata

$$\begin{aligned}
\mathtt{S}_0 &= \{\mathtt{parent}\} \\
S_1 &= \{\mathtt{delta_anc}(\mathtt{0}, \ldots)\} \\
S_2 &= \{\mathtt{all_anc}(\mathtt{0}, \ldots)\} \\
S_3 &= \{\mathtt{delta_anc}(\mathtt{1}, \ldots)\} \\
S_4 &= \{\mathtt{all_anc}(\mathtt{1}, \ldots)\} \\
&\ldots \\
S_{j\times 2+1} &= \{\mathtt{delta_anc}(\mathtt{j}, \ldots)\} \\
S_{j\times 2+2} &= \{\mathtt{all_anc}(\mathtt{j}, \ldots)\}
\end{aligned}$$

For an XY-stratified program P, the iterated fixpoint of Algorithm 9.2 becomes quite simple; basically it reduces to a repeated computation over the stratified program P_{bis}. However, since the temporal arguments have been removed from this program, we need to (1) store the temporal argument as an external fact $\mathtt{counter}(\mathtt{T})$, and (2) add a new goal $\mathtt{counter}(\mathtt{I}_r)$ to each exit rule r in P_{bis}, where $\mathtt{I}_r$ is the temporal argument in the original rule r. The program so constructed will be called the *synchronized* version of P_{bis}.

For instance, to obtain the synchronized version of the program in Example 10.11, we need to change the first rule to

$$\texttt{new\_delta\_anc}(\texttt{marc}) \leftarrow \quad \texttt{counter}(0).$$

since the temporal argument in the original exit rule was the constant 0.

Then, the iterated fixpoint computation for an XY-stratified program can be implemented by the following procedure:

Procedure 10.16 *Computing the well-founded model of an XY-stratified program P*

Inititialize: *Set* $T = 0$ *and insert the fact* counter(T).

Forever repeat the following two steps:

1. *Apply the iterated fixpoint computation to the synchronized program* P_{bis}*, and for each recursive predicate* q*, compute* new_q*. Return the* new_q *atoms so computed, after adding a temporal argument* T *to these atoms; the value of* T *is taken from* counter(T).
2. *For each recursive predicate* q*, replace* old_q *with* new_q*, computed in the previous step. Then, replace* counter(T) *with* counter(T + 1).

Thus, for Example 10.10, the goal counter(0) ensures that the exit rule only fires once immediately after the initialization step. However, we might have exit rules where the temporal argument is a constant greater than zero, or even is not a constant, and then the exit rules might produce results at later steps, too.

For the program in Example 10.4, Procedure 10.16 cannot terminate since it must compute all integers. However, for practical applications we need to have simple sufficient conditions that allow us to stop the computation. An effective condition can be formulated as follows: For each rule, there is at least one positive goal that cannot be satisfied for any value of its variables. Then, the following two conditions need to be checked: (i) the exit rules cannot produce any new values for values of T greater than the current one, and (ii) for each recursive rule, there is a positive goal, say, q, for which no new_q atoms were obtained at the last step. For Example 10.10, condition (i) is satisfied since the temporal argument is a constant; however, the third rule forever fails condition (ii), causing an infinite computation. Thus, the

programmer should add the goal `delta_anc(J, _)` to this rule. Once no new arcs are found, then because of the negated goal in the second rule, there is no `new_delta_anc(J, _)` atom and the computation stops.

The computation of Procedure 10.16 can be made very efficient by some simple improvements. The first improvement consists in observing that the replacement of `old_q` with `new_q` described in the last step of the procedure can become a zero-cost operation if properly implemented (e.g., by switching the reference pointers to the two relations). A second improvement concerns copy rules such as the last rule in Example 10.10. Observe that the body and the head of this rule are identical, except for the prefixes `new` or `old`, in its bi-state version (Example 10.11). Thus, in order to compute `new_all_anc`, we first execute the copy rule by simply setting the pointer to `new_all_anc` to point to `old_all_anc`—a zero-cost operation. The third rule is executed after that, since it can add tuples to `new_all_anc`.

In the next example, we use XY-stratified programs to compute temporal projections. Say, for instance, that we have a temporal relation as follows: `emp_dep_sal (Eno, Dept, Sal, From, To)`. Now, say that

```
emp_dep_sal(1001, shoe, 35000, 19920101, 19940101).
emp_dep_sal(1001, shoe, 36500, 19940101, 19960101).
```

represent two tuples from this relation. The first fact denotes that employee with `Eno` = `1001` has kept the same salary ($35,000) and department (`shoe`) from 1992/01/01 (year/month/day) till 1994/01/01.[4] According to the second fact, this employee, still in the `shoe` department, then received a salary of $36,500 from 1994/01/01 till 1996/01/01. If we now project out the salary and department information, these two intervals must be merged together. In this example, we have intervals overlapping over their endpoints. In more general situations, where a temporal projection eliminates some key attributes, we might have intervals overlapping each other over several time granules.

Thus, we use the program of Example 10.12, which iterates over two basic computation steps. The first step is defined by the `overlap` rule. This determines pairs of distinct overlapping intervals, where the first interval precedes (i.e., contains the start) of the second interval. The second step consists of deriving a new interval that begins at the start of the first interval, and ends at the later of the two endpoints. Finally, there is a copy rule that copies those intervals that do not overlap other intervals.

[4] In a Datalog system that supports the temporal data types discussed in Part II, those should be used instead of this crude representation.

This example uses the auxiliary predicates `distinct` and `select_larger`. The first verifies that two intervals are not the same interval. The second selects the larger of a pair of values. The program P of Example 10.12 is XY-stratified since the nodes of P_{bis} can be sorted into the following strata $\sigma_0 = \{$`distinct`, `select_larger`, `old_overlap`, `old_e_hist`$\}$, $\sigma_1 = \{$`new_overlap`$\}$, $\sigma_2 = \{$`new_e_hist`$\}$.

Example 10.12 Merging overlapping periods into maximal periods after a temporal projection

```
e_hist(0, Eno, Frm, To) ←          emp_dep_sal(0, Eno, D, S, Frm, To).

overlap(J + 1, Eno, Frm1, To1, Frm2, To2) ←
                                   e_hist(J, Eno, Frm1, To1),
                                   e_hist(J, Eno, Frm2, To2),
                                   Frm1 ≤ Frm2, Frm2 ≤ To1,
                                   distinct(Frm1, To1, Frm2, To2).

e_hist(J, Eno, Frm1, To) ←         overlap(J, Eno, Frm1, To1, Frm2, To2),
                                   select_larger(To1, To2, To).
e_hist(J + 1, Eno, Frm, To) ←      e_hist(J, Eno, Frm, To),
                                   overlap(J + 1, _, _, _, _, _),
                                   ¬overlap(J + 1, Eno, Frm, To, _, _),
                                   ¬overlap(J + 1, Eno, _, _, Frm, To).


final_e_hist(Eno, Frm, To) ←       e_hist(J, Eno, Frm, To),
                                   ¬e_hist(J + 1, _, _, _).


distinct(Frm1, To1, Frm2, To2) ←   To1 ≠ To2.
distinct(Frm1, To1, Frm2, To2) ←   Frm1 ≠ Frm2.

select_larger(X, Y, X) ←           X ≥ Y.
select_larger(X, Y, Y) ←           Y > X.
```

Thus, in the corresponding local stratification, the atoms `distinct` and the atoms `select_larger` go to the bottom stratum S_0. Then the atoms in σ_1 are in strata $S_{j\times 2+1}$, while those in σ_2 are now in strata $S_{j\times 2+2}$ (j denotes the temporal argument of these atoms).

The second `e_hist` rule in Example 10.12 is a qualified copy rule; that is, the head is copied from the body provided that certain conditions are satisfied. This can be implemented by letting `new_e_hist` and `old_e_hist`

share the same table, and use deletion-in-place for those tuples that do not satisfy the two negated goals. The goal `overlap(J, _, _, _, _)` ensures that the computation stops as soon as no more overlapping intervals are found.

As demonstrated by these examples, XY-stratified programs allow an efficient logic-based expression of procedural algorithms.

10.5 Updates and Active Rules

In general, logic-based systems have not dealt well with database updates. For instance, Prolog resorts to its operational semantics to give meaning to `assert` and `retract` operations. Similar problems are faced by deductive database systems, which, however, concentrate on changes in the base relations, rather than facts and rules as in Prolog. Active database rules contain updates in both their bodies and their heads. Much current research work pursues the objective of providing a unified treatment for active rules, dealing with updates, and deductive rules, dealing with queries. The two are now viewed as separate areas of database technology, although they often use similar techniques and concepts (see, for instance, the uses of stratification in Section 4.2). This section outlines a simple solution to these problems using Datalog_{1S} and XY-stratification.

The approach here proposed is driven by the threefold requirement of (1) providing a logical model for updates, (2) supporting the same queries that current deductive databases do, and (3) supporting the same rules that active databases currently do.

The first requirement can be satisfied by using history relations as extensional data. Thus, for each base relation R in the schema, there is a *history relation*, which keeps the history of all changes (updates) on R. For our university database, instead of having a `student` relation with facts of the following form:

```
student('Jim Black', cs, junior).
```

we have a `student_hist` relation containing the history of changes undergone by `Jim Black`'s record. Example 10.13 shows a possible history.

Example 10.13 The history of changes for `Jim Black`

```
student_hist(2301,+, 'Jim Black', ee, freshman).

student_hist(4007,-, 'Jim Black', ee, freshman).
student_hist(4007,+, 'Jim Black', ee, sophomore).
```

```
student_hist(4805,-, 'Jim Black', ee, sophomore).
student_hist(4805,+, 'Jim Black', cs, sophomore).

student_hist(6300,-, 'Jim Black', cs, sophomore).
student_hist(6300,+, 'Jim Black', cs, junior).
```

Thus, 'Jim Black' joined as a freshman in ee. The first column, 2301, is a change counter that is global for the system—that is, it is incremented for each change request. In fact, several changes can be made in the same SQL update statement: for instance, it might be that all the ee freshmen have been updated to sophomore in the same request. Thus, one year later, as Jim moves from freshman to sophomore, the change counter has been set to 4007. Therefore, there has been a total of $4,007 - 2,301$ database changes during the course of that year. Thus, we represent here deletions by the "$-$" sign in the second column and inserts by the "$+$" sign in the same column. An update is thus represented by a delete/insert pair having the same value in the temporal argument. Different representations for updates and other database events (e.g., representing the update directly, and timestamping the tuples) could also be handled in this modeling approach.

The remaining history of Jim Black's records states that he became a sophomore, and then he changed his major from ee to cs. Finally Jim became a junior—and that represents the current situation.

For a history database to be correct, it must satisfy a *continuity axiom*, which basically states that there is no jump in the change counters. This can be expressed through a predicate that registers when there is some change in some database relation. For instance, say that our university database, in addition to the tables

$$\texttt{student}(\texttt{Name}, \texttt{Major}, \texttt{Year});\ \texttt{took}(\texttt{Name}, \texttt{Course}, \texttt{Grade})$$

discussed in Chapter 8, also contains the relation

$$\texttt{alumni}(\texttt{Name}, \texttt{Sex}, \texttt{Degree}, \texttt{ClassOf})$$

which obviously records the alumni who graduated from college in the previous years. Then we will need three rules to keep track of all changes:

$$\begin{aligned}
\texttt{change}(\texttt{J}) \leftarrow\ & \texttt{student_hist}(\texttt{J}, _, _, _, _).\\
\texttt{change}(\texttt{J}) \leftarrow\ & \texttt{took_hist}(\texttt{J}, _, _, _, _).\\
\texttt{change}(\texttt{J}) \leftarrow\ & \texttt{alumni_hist}(\texttt{J}, _, _, _, _, _).
\end{aligned}$$

Thus, there is a rule for each history relation. A violation to the continuity axiom can be expressed as follows:

$$\texttt{bad\_history} \leftarrow \quad \texttt{change}(\texttt{J}+1), \neg\texttt{change}(\texttt{J}).$$

Let us turn now to the second requirement: the support of deductive queries against the current database. This is achieved through snapshot predicates, derived from the history relations using frame axioms.

For the `student` relation, for instance, we have the following rules:

Example 10.14 Snapshot predicates for `student` via frame axioms

$$\begin{aligned}
&\texttt{student_snap}(\texttt{J}+1, \texttt{Name}, \texttt{Major}, \texttt{Level}) \leftarrow \\
&\qquad\qquad \texttt{student_snap}(\texttt{J}, \texttt{Name}, \texttt{Major}, \texttt{Level}), \\
&\qquad\qquad \neg\texttt{student_hist}(\texttt{J}+1, -, \texttt{Name}, \texttt{Major}, \texttt{Level}). \\
&\texttt{student_snap}(\texttt{J}, \texttt{Name}, \texttt{Major}, \texttt{Level}) \leftarrow \\
&\qquad\qquad \texttt{student_hist}(\texttt{J}, +, \texttt{Name}, \texttt{Major}, \texttt{Level}).
\end{aligned}$$

These rules express what are commonly known as *frame axioms*. Basically, the content of a database relation after some change is the same as that in the previous state, minus the deleted tuples, and plus the inserted tuples. Observe that the recursive `student_snap` rules so obtained are XY-stratified. Furthermore, observe the first rule in Example 10.14, which computes the effects of deletions. This is a copy rule and, therefore, it will be implemented by removing the tuples satisfying the negated goal (i.e., the deleted tuples) and leaving the rest of the relation unchanged. Similar frame axiom rules will be defined for each relation in the schema.

Deductive queries are then answered against the current content of the database (i.e., the final value of the change counter).

Example 10.15 The current content of the relation `student`

$$\begin{aligned}
&\texttt{current_state}(\texttt{J}) \leftarrow && \texttt{change}(\texttt{J}), \neg\texttt{change}(\texttt{J}+1). \\
&\texttt{student}(\texttt{Name}, \texttt{Major}, \texttt{Year}) \leftarrow && \texttt{student_snap}(\texttt{J}, \texttt{Name}, \texttt{Major}, \texttt{Year}), \\
&&& \texttt{current_state}(\texttt{J}).
\end{aligned}$$

Because of the continuity axiom, $\texttt{change}(\texttt{J}), \neg\texttt{change}(\texttt{J}+1)$ defines the current state `J`. Similar rules will be written for the remaining relations in the database. The predicates so derived are then used in deductive rules described in the previous chapters.

We can now turn to the problem of modeling active rules. Here we limit our discussion to *Event-Condition-Action* rules, under the immediate-after activation semantics, that were discussed in Chapter 2. These can be modeled naturally in this framework, since conditions can be expressed against the snapshot relations, while events and actions can be expressed against history relations. For instance, say that we have the following active rules:

A_1 : *If a student is added to the alumni relation, then delete his name from the student relation, provided that this is a senior-level student.*

A_2 : *If a person takes a course, and the name of this person is not in the student relation, then add that name to the student relation, using the (null) value* `tba` *for Major and Level.*

Under the "immediately after" activation semantics, discussed in Section 2.2, these rules can be modeled as follows:

Example 10.16 Active rules on histories and snapshots

$$A_1 : \texttt{student\_hist}(\texttt{J}+1, -, \texttt{Name}, \texttt{Major}, \texttt{senior}) \leftarrow \texttt{alumni\_hist}(\texttt{J}, +, \texttt{Name}, \_, \_, \_), \texttt{student\_snap}(\texttt{J}, \texttt{Name}, \texttt{Major}, \texttt{senior}).$$

$$A_2 : \texttt{student\_hist}(\texttt{J}+1, +, \texttt{Name}, \texttt{tba}, \texttt{tba}) \leftarrow \texttt{took\_hist}(\texttt{J}, +, \texttt{Name}, \_, \_), \neg\texttt{student\_snap}(\texttt{J}, \texttt{Name}, \_, \_).$$

Let $\mathcal{A}$ be an active program, that is, the logic program consisting of (1) the history relations, (2) the change predicates, (3) the snapshot predicates, and (4) the active rules. Because of its XY-stratified structure, this program has a unique stable model M, which defines the meaning of the program.

Now assume that a new tuple is added to the history relation. Active rules might trigger changes and then tuples are added to the history relations until no more rules can fire. However, only the external changes requested by a user, and those triggered by active rules, can be in the history relations. Therefore, an active database $\mathcal{A}$ must satisfy the following two axioms:

1. *Completeness Axiom.* The history relations in $\mathcal{A}$ must be identical to the history relations in the stable model of $\mathcal{A}$.

2. *External Causation Axiom.* Let $\mathcal{A}_{ext}$ be the logic program obtained from $\mathcal{A}$ by eliminating from the history relations all changes but the external changes requested by users. Then, the stable model of $\mathcal{A}_{ext}$ and the stable model of the original $\mathcal{A}$ must be identical.

Thus, while predicates defined by deductive rules behave as virtual views, the history relations behave as concrete views. Whenever a new change is requested by users, all changes implied by the active rules are also added to the history relations. The complete set of new entries added to the history relations define the response of the system to the changes requested by the user.

10.6 Nondeterministic Reasoning

Say that, with relation `student(Name, Major, Year)`, our university database contains the relation `professor(Name, Major)`. In fact, say that our toy database contains only the following facts:

```
student('Jim Black', ee, senior).
professor(ohm, ee).
professor(bell, ee).
```

Now, the rule is that the major of a student must match his/her advisor's major area of specialization. Then eligible advisors can be computed as follows:

```
elig_adv(S, P) ← student(S, Major, Year), professor(P, Major).
```

This yields

```
elig_adv('Jim Black', ohm).
elig_adv('Jim Black', bell).
```

But, since a student can only have one advisor, the goal `choice((S), (P))` must be added to force the selection of a unique advisor, out of the eligible advisors, for a student.

Example 10.17 Computation of unique advisors by choice rules

```
actual_adv(S, P) ← student(S, Major, Levl), professor(P, Major),
                   choice((S), (P)).
```

The goal `choice((S),(P))` can also be viewed as enforcing a functional dependency (FD) `S` $\rightarrow$ `P`; thus, in `actual_adv`, the second column (professor name) is functionally dependent on the first one (student name).

The result of executing this rule is *nondeterministic*. It can either give a singleton relation containing the tuple ('`Jim Black`', `ohm`) or that containing the tuple ('`Jim Black`', `bell`).

A program where the rules contain choice goals is called a *choice program*. The semantics of a choice program P can be defined by transforming P into a program with negation, $SV(P)$, called the *stable version* of a choice program P. $SV(P)$ exhibits a multiplicity of stable models, each obeying the FDs defined by the choice goals. Each stable model for $SV(P)$ corresponds to an alternative set of answers for P and is called a *choice model* for P. $SV(P)$ is defined as follows:

Definition 10.17 *The* stable version $SV(P)$ *of a choice program P is obtained by the following transformation. Consider a choice rule r in P:*

$$r : A \leftarrow B(Z),\ choice((X_1),(Y_1)),\ \ldots,\ choice((X_k),(Y_k)).$$

where,

(i) $B(Z)$ denotes the conjunction of all the choice goals of r that are not choice goals, and

(ii) $X_i,\ Y_i,\ Z$, $1 \leq i \leq k$, denote vectors of variables occurring in the body of r such that $X_i \cap Y_i = \emptyset$ and $X_i, Y_i \subseteq Z$.

Then the original program P is transformed as follows:

1. *Replace r with a rule r' obtained by substituting the choice goals with the atom $chosen_r(W)$:*

$$r' : A \leftarrow B(Z),\ chosen_r(W).$$

where $W \subseteq Z$ is the list of all variables appearing in choice goals, i.e., $W = \bigcup_{1 \leq j \leq k} X_j \cup Y_j$.

2. *Add the new rule*

$$chosen_r(W) \leftarrow B(Z),\ \neg diffChoice_r(W).$$

3. *For each choice atom $choice((X_i),(Y_i))$ $(1 \leq i \leq k)$, add the new rule*

$$diffChoice_r(W) \leftarrow chosen_r(W'),\ Y_i \neq Y_i'.$$

where (i) the list of variables W' is derived from W by replacing each $A \in Y_i$ with a new variable $A' \in Y_i'$ (i.e., by priming those variables), and (ii) $Y_i \neq Y_i'$ is true if $A \neq A'$, for some variable $A \in Y_i$ and its primed counterpart $A' \in Y_i'$.

The stable version of Example 10.17 is given in Example 10.18, which can be read as a statement that a professor will be assigned to each student if a different professor has not been assigned to the same student.

Example 10.18 The stable version of the rule in Example 10.17

```
actual_adv(S, P) ←   student(S, Majr, Yr), professor(P, Majr),
                     chosen(S, P).
chosen(S, P) ←       student(S, Majr, Yr), professor(P, Majr),
                     ¬diffChoice(S, P).
diffChoice(S, P) ←   chosen(S, P'), P ≠ P'.
```

In general, the program $SV(P)$ generated by the transformation discussed above has the following properties:

- $SV(P)$ has one or more total stable models.
- The `chosen` atoms in each stable model of $SV(P)$ obey the FDs defined by the choice goals.

The stable models of $SV(P)$ are called *choice models* for P.

Stratified Datalog programs with choice are in DB-PTIME: actually they can be implemented efficiently by producing `chosen` atoms one at a time and memorizing them in a table. The `diffchoice` atoms need not be computed and stored; rather, the goal ¬`diffchoice` can simply be checked dynamically against the table `chosen`.

The use of choice is critical in many applications. For instance, the following nonrecursive rules can be used to determine whether there are more boys than girls in a database containing the unary relations `boy` and `girl`:

Example 10.19 Are there more boys than girls in our database?

```
match(Bname, Gname) ←   boy(Bname), girl(Gname).
                        choice((Bname), (Gname)),
                        choice((Gname), (Bname)).
matched_boy(Bname) ←    match(Bname, Gname).
moreboys ←              boy(Bname), ¬matched_boy(Bname).
```

The most significant applications of choice involve the use of choice in recursive predicates. For instance, the following program computes the spanning tree, starting from the source node a, for a graph where an arc from node b to d is represented by the database fact g(b, d).

Example 10.20 Computing a spanning tree

```
st(root, a).
st(X, Y) ← st(_, X), g(X, Y), Y ≠ a, choice((Y), (X)).
```

In this example, the goal Y ≠ a ensures that, in st, the end-node for the arc produced by the exit rule has an in-degree of one; likewise, the goal choice((Y), (X)) ensures that the end-nodes for the arcs generated by the recursive rule have an in-degree of one.

Stratified Datalog programs with choice are also DB-PTIME complete, without having to assume that the universe is totally ordered. Indeed, the following program defines a total order for the elements of a set d(X) by constructing an immediate-successor relation for its elements (root is a distinguished new symbol):

Example 10.21 Ordering a domain

```
ordered_d(root, root).
ordered_d(X, Y) ← ordered_d(_, X), d(Y),
                  choice((X), (Y)), choice((Y), (X)).
```

The choice goals in Example 10.21 ensure that once an arc (X, Y) is generated, this is the only arc leaving the source node X and the only arc entering the sink node Y.

Set aggregates on the elements of the set d(X), including the parity query, are easily computed using the relation ordered_d. Alternatively, these aggregates can be computed directly using a program similar to that of Example 10.21. For instance, the sum of all elements in d(X) can be computed as follows:

Example 10.22 The sum of the elements in d(X)

```
sum_d(root, root, 0) ← d(_).
sum_d(X, Y, SY) ← sum_d(_, X, SX), d(Y),
                  choice((X), (Y)), choice((Y), (X)), SY = SX + Y.
total_d(Sum) ← sum_d(_, X, Sum), ¬sum_d(X, _, _).
```

If we eliminate the choice goal from the program P of Example 10.22, we obtain a program P' that is stratified with respect to negation. Therefore, the original P is called a *stratified choice program*. Stratified choice programs always have stable models. Moreover, these stable models can be computed by an iterated fixpoint computation that computes choice models for strata where rules have choice goals. For instance, in Example 10.22, the choice model of the `sum_d` stratum is computed first, and the next stratum containing `total_d` is computed after that. Likewise, choice goals can be added to XY-stratified programs, yielding programs that have multiple stable models. These models can be computed by Procedure 10.16, which is now iterating over stratified choice programs.

The choice construct is significant for both nondeterministic and deterministic queries. A nondeterministic query is one where any answer out of a set is acceptable. This is, for instance, the case of Example 10.17, where an advisor has to be assigned to a student. Example 10.22 illustrates the use of choice to compute a deterministic query; in fact, the sum of the elements of the set is independent from the order in which these are visited.

Since stratified Datalog on an ordered domain is DB-PTIME complete, Example 10.21 ensures that stratified choice programs are also DB-PTIME complete for deterministic queries. Furthermore, the use of choice, unlike the assumption of an underlying ordered domain, also ensures the *genericity* of queries. Basically, a query is said to be generic if it is independent of permutations of the order of constants in the database.

When choice is used in the computation of stratified or XY-stratified programs, no previously made choice needs to be repudiated later, since the iterated fixpoint always generates a stable model. This produces "don't care" nondeterminism, generating polynomial-time computations.

In the more general case, however, finding a stable model for a program requires the use of intelligent (oracle-type) nondeterminism; in practice, this can only be realized by an exponential search. For instance, the following program determines if there exists a Hamiltonian path in a graph. A graph has a Hamiltonian path iff there is a simple path that visits all nodes exactly once.

Example 10.23 Hamiltonian path in a directed graph g(X, Y)

```
simplepath(root, root).
simplepath(X, Y) ← simple_path(_, X), g(X, Y),
                   choice((X), (Y)), choice((Y), (X)).
nonhppath ← n(X), ¬simplepath(_, X).
q ← ¬q, nonhppath.
```

Let M be a model for this program. If `nonhppath` is true in M, then rule $\texttt{q} \leftarrow \neg\texttt{q}$ must also be satisfied by M. Thus, M cannot be a stable model. Thus, this program has a stable model iff there exists a Hamiltonian path. Thus, searching for a stable model of this program might require all alternative choices to be explored for each possible node (e.g., using backtracking). This generates an exponential computation. In fact, since the Hamiltonian path problem is known to be $\mathcal{NP}$-complete, the stable version of Example 10.23 provides a simple proof that deciding whether a stable model exists for a program is $\mathcal{NP}$-hard.

10.7 Research Directions

The main focus of current research is breaking through the many barriers that have limited the application of logic-oriented databases in various domains. Developing a logic-based theory of database changes and events and supporting the integration of active and deductive databases represent two important objectives in this effort. In contrast with the simple approach described in this chapter, many of the approaches proposed in the past have been very sophisticated and complex. A major line of AI research uses situation calculus, where function symbols are used to represent changes and histories. In addition to modeling database updates, this approach has been used for integrity constraints, AI planning, reasoning with hypothetical updates, and other difficult problems. A logic-based framework for modeling changes and frame axioms in situation calculus can follow an open-world approach with explicit-negation, or take the implicit-negation approach, often in conjunction with stable model semantics. The transaction logic approach developed by Bonner and Kifer recasts some of this power and complexity into a path-based formalism (Datalog_{1S} is state-oriented) attuned to database transactions.

Support for temporal and spatial reasoning in databases represents a promising area of current research, and so is the topic of reasoning with uncertainty, discussed in Part V. These lines of research often tackle issues of partial order, since nonmonotonic programs written to reason with time or uncertainty, for example, can become monotonic once the appropriate lattice-based representation is found. A closely related research line investigates the problem of monotonic aggregation in databases.

In the previous chapter we discussed some of the technical challenges facing the design of deductive database systems and of recursive extensions for SQL3. It is reasonable to expect that the next generation of system prototypes will resolve those problems and move toward the integration of

the active, deductive, and temporal capabilities previously outlined. Furthermore, object-oriented features will be included in these systems. The integration of deductive and object-oriented databases has been a topic of active research and will remain so for the near future.

In the end, future developments in the field will be driven more by applications than by technological advances. In particular, the uses of deductive systems to realize distributed mediators and to perform knowledge discovery from databases are emerging as very promising application domains for the technology.

10.8 Bibliographic Notes

The CWA was proposed by Reiter [354] and was later generalized by Minker [288]. The notion of stratification has seen several independent discoveries, including those by Chandra and Harel [101], Apt, Blair and Walker [20], Naqvi [301], Van Gelder [442], and Przymusinski [341], who is also credited with the notion of locally stratified programs. The undecidability of this class of program was proven in [61]. A study of SLD-NF and its limitations can be found in [109] and [269]. Well-founded models (partial and total) were introduced by Van Gelder, Ross, and Schlipf, using the notion of unfounded sets [444]; the alternating fixpoint is due to Van Gelder [443]. The notion of modular stratification is a significant step toward practical realizations of well-founded semantics [364]. Implementations of this semantics in the top-down framework were also studied in [103].

The notion of stable models is due to Gelfond and Lifschitz [186]. This concept distills in a simple definition many AI ideas following McCarthy's notion of circumscription [279]. The definition of stable models can also be generalized to partial models, which exist for every program but contain undefined atoms. Three-valued logic provides a convenient tool to define partial stable models [342], although two-valued logic is quite up to the task [376]. An in-depth treatment of nonmonotonic formalisms, including defaults, can be found in [276].

A survey of query languages based on temporal logic can be found in [107]. Different kinds of temporal logic, including PLTL, are surveyed in [357] and [142]. Temporal applications of Datalog_{1S} and its underlying theory were elucidated by Chomicki [106, 46].

The notion of XY-stratification is due to Zaniolo, Arni, and Ong [469]. The related concept of explicitly locally stratified programs was investigated in [241]. The use of XY-stratified programs to model updates and active rules was explored in [466, 467]. Brogi, Subrahmanian, and Zaniolo extended XY-

stratification with the choice construct to model various planning problems [80]. The logic of actions has been the focus of a large body of research; the use of situation calculus and classical logic was explored by Reiter [339, 354]. Among the many approaches using implicit negation, we will only mention [33, 187, 34].

Transaction logic is described in [68]. F-logic represents a leading proposal in the area of deductive and object-oriented databases [243]. An introduction to the problem of monotonic aggregation can be found in [365].

The concept of choice, first proposed by Krishnamurthy and Naqvi [257], was then revised by Saccà and Zaniolo [375] using the stable-model semantics; the concept of dynamic choice was introduced by Giannotti, Pedreschi, Saccá and Zaniolo in [189], and its expressive power was studied in [123]. Other nondeterministic extensions to Datalog languages were investigated by Abiteboul and Vianu [4].

10.9 Exercises

10.1. Let P be a Datalog program. Show that $\Gamma_P^{\uparrow\omega}$ can be computed in time that is polynomial in the size of P's Herbrand base.

10.2. Use Example 10.23 and the results from Exercise 10.1 to show that deciding whether stable models exist for a given program is $\mathcal{NP}$-complete.

10.3. Consider the program

$$\begin{aligned} \mathtt{c} &\leftarrow \mathtt{a}, \neg\mathtt{c}. \\ \mathtt{a} &\leftarrow \neg\mathtt{b}. \\ \mathtt{b} &\leftarrow \mathtt{a}, \neg\mathtt{c}. \end{aligned}$$

For this program, write all models, minimal models, fixpoints, minimal fixpoints, and stable models, if any. Also give $T_P^{\uparrow\omega}$.

10.4. Prove that every stable model is a minimal model.

10.5. Prove that every stable model for P is a minimal fixpoint for T_P.

10.6. Prove that the program in Example 10.5 is not locally stratified.

10.7. Explain how the alternating fixpoint computes the stable model for a stratified program. Perform the computation on Example 8.7.

10.8. The until ($\mathcal{U}$) operator of PLTL is monotonic; give a positive program that expresses this operator.

10.9. Write an XY-stratified program for the BoM Example 8.19, where the computation of longest time required for all subparts of a given part is performed in the recursive rules. What are the local strata for the resulting program?

10.10. Given a directed graph `g(a, b, d)`, where `a` and `b` are nodes, and `d` is their distance, determine

a. the least distances of a given node to all other nodes of the graph, and

b. the distances between all node pairs.

Express various algorithms for computing such distances by XY-stratified programs.

10.11. For the active program $\mathcal{A}$ described in Section 10.5, derive A_{bis} and its synchronized version. Explain why the program is XY-stratified and how the actual computation will take place.

10.12. Given the relation `club_member(Name, Sex)`, write a nonrecursive Datalog program to determine whether there is exactly the same number of males and females in the club. Use choice but not function symbols.

10.13. Use choice to write a recursive program that takes a set of database predicates `b(X)` and produces a relation `sb(X, I)`, with `I` a unique sequence number attached to each `X`.

10.14. Express the parity query using choice (do not assume a total order of the universe).

10.15. Write a program with choice and stratified negation that performs the depth-first traversal of a directed graph.

Author Index