PART

D Datalog and Recursion

In Part B, we considered query languages ranging from conjunctive queries to first-order queries in the three paradigms: algebraic, logic, and deductive. We did this by enriching the conjunctive queries first with union (disjunction) and then with difference (negation). In this part, we further enrich these languages by adding *recursion*. First we add recursion to the conjunctive queries, which yields *datalog*. We study this language in Chapter 12. Although it is too limited for practical use, datalog illustrates some of the essential aspects of recursion. Furthermore, most existing optimization techniques have been developed for datalog.

Datalog owes a great debt to Prolog and the logic-programming area in general. A fundamental contribution of the logic-programming paradigm to relational query languages is its elegant notation for expressing recursion. The perspective of databases, however, is significantly different from that of logic programming. (For example, in databases datalog programs define mappings from instances to instances, whereas logic programs generally carry their data with them and are studied as stand-alone entities.) We adapt the logic-programming approach to the framework of databases.

We study evaluation techniques for datalog programs in Chapter 13, which covers the main optimization techniques developed for recursion in query languages, including seminaive evaluation and magic sets.

Although datalog is of great theoretical importance, it is not adequate as a practical query language because of the lack of negation. In particular, it cannot express even the first-order queries. Chapters 14 and 15 deal with languages combining recursion and negation, which are proper extensions of first-order queries. Chapter 14 considers the issue of combining negation and recursion. Languages are presented from all three paradigms, which support both negation and recursion. The semantics of each one is defined in fundamentally operational terms, which include datalog with negation and a straightforward, fixpoint semantics. As will be seen, the elegant correspondence between languages in the three paradigms is maintained in the presence of recursion.

Chapter 15 considers approaches to incorporating negation in datalog that are closer in spirit to logic programming. Several important semantics for negation are presented, including stratification and well-founded semantics.

12 Datalog

Alice: *What do we see next?*
Riccardo: *We introduce recursion.*
Sergio: *He means we ask queries about your ancestors.*
Alice: *Are you leading me down a garden path?*
Vittorio: *Kind of—queries related to paths in a graph call for recursion and are crucial for many applications.*

For a long time, relational calculus and algebra were considered *the* database languages. Codd even defined as "complete" a language that would yield precisely relational calculus. Nonetheless, there are simple operations on data that cannot be realized in the calculus. The most conspicuous example is graph transitive closure. In this chapter, we study a language that captures such queries and is thus more "complete" than relational calculus.[1] The language, called datalog, provides a feature not encountered in languages studied so far: *recursion*.

We start with an example that motivates the need for recursion. Consider a database for the Parisian **Metro**. Note that this database essentially describes a graph. (Database applications in which part of the data is a graph are common.) To avoid making the **Metro** database too static, we assume that the database is describing the available metro connections on a day of strike (not an unusual occurrence). So some connections may be missing, and the graph may be partitioned. An instance of this database is shown in Fig. 12.1.

Natural queries to ask are as follows:

(12.1) What are the stations reachable from Odeon?

(12.2) What lines can be reached from Odeon?

(12.3) Can we go from Odeon to Chatelet?

(12.4) Are all pairs of stations connected?

(12.5) Is there a cycle in the graph (i.e., a station reachable in one or more stops from itself)?

Unfortunately, such queries cannot be answered in the calculus without using some *a*

[1] We postpone a serious discussion of completeness until Part E, where we tackle fundamental issues such as "What is a formal definition of data manipulation (as opposed to arbitrary computation)? What is a reasonable definition of completeness for database languages?"

Links

Line	*Station*	*Next Station*
4	*St.-Germain*	*Odeon*
4	*Odeon*	*St.-Michel*
4	*St.-Michel*	*Chatelet*
1	*Chatelet*	*Louvre*
1	*Louvre*	*Palais-Royal*
1	*Palais-Royal*	*Tuileries*
1	*Tuileries*	*Concorde*
9	*Pont de Sevres*	*Billancourt*
9	*Billancourt*	*Michel-Ange*
9	*Michel-Ange*	*Iena*
9	*Iena*	*F. D. Roosevelt*
9	*F. D. Roosevelt*	*Republique*
9	*Republique*	*Voltaire*

Figure 12.1: An instance I of the **Metro** database

priori knowledge on the **Metro** graph, such as the graph diameter. More generally, given a graph G, a particular vertex a, and an integer n, it is easy to write a calculus query finding the vertexes at distance less than n from a; but it seems difficult to find a query for all vertexes reachable from a, regardless of the distance. We will prove formally in Chapter 17 that such a query is *not* expressible in the calculus. Intuitively, the reason is the lack of recursion in the calculus.

The objective of this chapter is to extend some of the database languages considered so far with recursion. Although there are many ways to do this (see also Chapter 14), we focus in this chapter on an approach inspired by logic programming. This leads to a field called deductive databases, or database logic programming, which shares motivation and techniques with the logic-programming area.

Most of the activity in deductive databases has focused on a toy language called *datalog*, which extends the conjunctive queries with recursion. The interaction between negation and recursion is more tricky and is considered in Chapters 14 and 15. The importance of datalog for deductive databases is analogous to that of the conjunctive queries for the relational model. Most optimization techniques for relational algebra were developed for conjunctive queries. Similarly, in this chapter most of the optimization techniques in deductive databases have been developed around datalog (see Chapter 13).

Before formally presenting the language datalog, we present informally the syntax and various semantics that are considered for that language. Following is a datalog program P_{TC} that computes the transitive closure of a graph. The graph is represented in relation G and its transitive closure in relation T:

$$T(x, y) \leftarrow G(x, y)$$
$$T(x, y) \leftarrow G(x, z), T(z, y).$$

Observe that, except for the fact that relation T occurs both in the head and body of the second rule, these look like the nonrecursive datalog rules of Chapter 4.

A datalog program defines the relations that occur in heads of rules based on other relations. The definition is recursive, so defined relations can also occur in bodies of rules. Thus a datalog program is interpreted as a mapping from instances over the relations occurring in the bodies only, to instances over the relations occurring in the heads. For instance, the preceding program maps a relation over G (a graph) to a relation over T (its transitive closure).

A surprising and elegant property of datalog, and of logic programming in general, is that there are three very different but equivalent approaches to defining the semantics. We present the three approaches informally now.

A first approach is *model theoretic*. We view the rules as logical sentences stating a property of the desired result. For instance, the preceding rules yield the logical formulas

$$\forall x, y(T(x, y) \leftarrow G(x, y)) \tag{1}$$

$$\forall x, y, z(T(x, y) \leftarrow (G(x, z) \wedge T(z, y))). \tag{2}$$

The result T must satisfy the foregoing sentences. However, this is not sufficient to determine the result uniquely because it is easy to see that there are many Ts that satisfy the sentences. However, it turns out that the result becomes unique if one adds the following natural minimality requirement: T consists of the smallest set of facts that makes the sentences true. As it turns out, for each datalog program and input, there is a unique minimal model. This defines the semantics of a datalog program. For example, suppose that the instance contains

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

It turns out that $T(a, d)$ holds in each instance that obeys (1) and (2) and where these three facts hold. In particular, it belongs to the minimum model of (1) and (2).

The second *proof-theoretic* approach is based on obtaining proofs of facts. A proof of the fact $T(a, d)$ is as follows:

(i) $G(c, d)$ belongs to the instance;

(ii) $T(c, d)$ using (i) and the first rule;

(iii) $G(b, c)$ belongs to the instance;

(iv) $T(b, d)$ using (iii), (ii), and the second rule;

(v) $G(a, b)$ belongs to the instance;

(vi) $T(a, d)$ using (v), (iv), and the second rule.

A fact is in the result if there exists a proof for it using the rules and the database facts.

In the proof-theoretic perspective, there are two ways to derive facts. The first is to view programs as "factories" producing all facts that can be proven from known facts. The rules are then used *bottom up*, starting from the known facts and deriving all possible new facts. An alternative *top-down* evaluation starts from a fact to be proven and attempts to demonstrate it by deriving lemmas that are needed for the proof. This is the underlying

intuition of a particular technique (called resolution) that originated in the theorem-proving field and lies at the core of the logic-programming area.

As an example of the top-down approach, suppose that we wish to prove $T(a, d)$. Then by the second rule, this can be done by proving $G(a, b)$ and $T(b, d)$. We know $G(a, b)$, a database fact. We are thus left with proving $T(b, d)$. By the second rule again, it suffices to prove $G(b, c)$ (a database fact) and $T(c, d)$. This last fact can be proven using the first rule. Observe that this yields the foregoing proof (i) to (vi). Resolution is thus a particular technique for obtaining such proofs. As detailed later, resolution permits variables as well as values in the goals to be proven and the steps used in the proof.

The last approach is the *fixpoint* approach. We will see that the semantics of the program can be defined as a particular solution of a fixpoint equation. This approach leads to iterating a query until a fixpoint is reached and is thus procedural in nature. However, this computes again the facts that can be deduced by applications of the rules, and in that respect it is tightly connected to the (bottom-up) proof-theoretic approach. It corresponds to a natural strategy for generating proofs where shorter proofs are produced before longer proofs so facts are proven "as soon as possible."

In the next sections we describe in more detail the syntax, model-theoretic, fixpoint, and proof-theoretic semantics of datalog. As a rule, we introduce only the minimum amount of terminology from logic programming needed in the special database case. However, we make brief excursions into the wider framework in the text and exercises. The last section deals with static analysis of datalog programs. It provides decidability and undecidability results for several fundamental properties of programs. Techniques for the evaluation of datalog programs are discussed separately in Chapter 13.

12.1 Syntax of Datalog

As mentioned earlier, the syntax of datalog is similar to that of languages introduced in Chapter 4. It is an extension of nonrecursive datalog, which was introduced in Chapter 4. We provide next a detailed definition of its syntax. We also briefly introduce some of the fundamental differences between datalog and logic programming.

DEFINITION 12.1.1 A *(datalog) rule* is an expression of the form

$$R_1(u_1) \leftarrow R_2(u_2), \ldots, R_n(u_n),$$

where $n \geq 1$, $R_1, \ldots, R_n$ are relation names and $u_1, \ldots, u_n$ are free tuples of appropriate arities. Each variable occurring in u_1 must occur in at least one of $u_2, \ldots, u_n$. A *datalog program* is a finite set of datalog rules.

The *head* of the rule is the expression $R_1(u_1)$; and $R_2(u_2), \ldots, R_n(u_n)$ forms the *body*.

The set of constants occurring in a datalog program P is denoted $adom(P)$; and for an instance $\mathbf{I}$, we use $adom(P, \mathbf{I})$ as an abbreviation for $adom(P) \cup adom(\mathbf{I})$.

We next recall a definition from Chapter 4 that is central to this chapter.

Definition 12.1.2 Given a valuation ν, an *instantiation*

$$R_1(\nu(u_1)) \leftarrow R_2(\nu(u_2)), \ldots, R_n(\nu(u_n))$$

of a rule $R_1(u_1) \leftarrow R_2(u_2), \ldots, R_n(u_n)$ with ν is obtained by replacing each variable x by $\nu(x)$.

Let P be a datalog program. An *extensional* relation is a relation occurring only in the body of the rules. An *intensional* relation is a relation occurring in the head of some rule of P. The *extensional (database) schema*, denoted $edb(P)$, consists of the set of all extensional relation names; whereas the *intensional schema* $idb(P)$ consists of all the intensional ones. The *schema* of P, denoted $sch(P)$, is the union of $edb(P)$ and $idb(P)$. The semantics of a datalog program is a mapping from database instances over $edb(P)$ to database instances over $idb(P)$. In some contexts, we call the input data the extensional database and the program the intensional database. Note also that in the context of logic-based languages, the term *predicate* is often used in place of the term *relation name*.

Let us consider an example.

Example 12.1.3 The following program P_{metro} computes the answers to queries (12.1), (12.2), and (12.3):

$$\begin{array}{lcl}
St_Reachable(x, x) & \leftarrow & \\
St_Reachable(x, y) & \leftarrow & St_Reachable(x, z), Links(u, z, y) \\
Li_Reachable(x, u) & \leftarrow & St_Reachable(x, z), Links(u, z, y) \\
Ans_1(y) & \leftarrow & St_Reachable(Odeon, y) \\
Ans_2(u) & \leftarrow & Li_Reachable(Odeon, u) \\
Ans_3() & \leftarrow & St_Reachable(Odeon, Chatelet)
\end{array}$$

Observe that *St_Reachable* is defined using recursion. Clearly,

$$\begin{aligned}
edb(P_{metro}) &= \{Links\}, \\
idb(P_{metro}) &= \{St_Reachable, Li_Reachable, Ans_1, Ans_2, Ans_3\}
\end{aligned}$$

For example, an instantiation of the second rule of P_{metro} is as follows:

$$St\_Reachable(Odeon, Louvre) \leftarrow St\_Reachable(Odeon, Chatelet), Links(1, Chatelet, Louvre)$$

Datalog versus Logic Programming

Given the close correspondence between datalog and logic programming, we briefly highlight the central differences between these two fields. The major difference is that logic programming permits function symbols, but datalog does not.

EXAMPLE 12.1.4 The simple logic program P_{leq} is given by

$$\begin{aligned} leq(0, x) &\leftarrow \\ leq(s(x), s(y)) &\leftarrow leq(x, y) \\ leq(x, +(x, y)) &\leftarrow \\ leq(x, z) &\leftarrow leq(x, y), leq(y, z) \end{aligned}$$

Here 0 is a constant, s a unary function sysmbol, $+$ a binary function sysmbol, and *leq* a binary predicate. Intuitively, s might be viewed as the successor function, $+$ as addition, and *leq* as capturing the less-than-or-equal relation. However, in logic programming the function symbols are given the "free" interpretation—two terms are considered nonequal whenever they are syntactically different. For example, the terms $+(0, s(0)), +(s(0), 0)$, and $s(0)$ are all nonequal. Importantly, functional terms can be used in logic programming to represent intricate data structures, such as lists and trees.

Observe also that in the preceding program the variable x occurs in the head of the first rule and not in the body, and analogously for the third rule.

Another important difference between deductive databases and logic programs concerns perspectives on how they are typically used. In databases it is assumed that the database is relatively large and the number of rules relatively small. Furthermore, a datalog program P is typically viewed as defining a mapping from instances over the *edb* to instances over the *idb*. In logic programming the focus is different. It is generally assumed that the base data is incorporated directly into the program. For example, in logic programming the contents of instance *Link* in the **Metro** database would be represented using rules such as $Link(4, St.\text{-}Germain, Odeon) \leftarrow$. Thus if the base data changes, the logic program itself is changed. Another distinction, mentioned in the preceding example, is that logic programs can construct and manipulate complex data structures encoded by terms involving function symbols.

Later in this chapter we present further comparisons of the two frameworks.

12.2 Model-Theoretic Semantics

The key idea of the model-theoretic approach is to view the program as a set of first-order sentences (also called a first-order theory) that describes the desired answer. Thus the database instance constituting the result satisfies the sentences. Such an instance is also called a *model* of the sentences. However, there can be many (indeed, infinitely many) instances satisfying the sentences of a program. Thus the sentences themselves do not uniquely identify the answer; it is necessary to specify which of the models is

the *intended* answer. This is usually done based on assumptions that are external to the sentences themselves. In this section we formalize (1) the relationship between rules and logical sentences, (2) the notion of model, and (3) the concept of intended model.

We begin by associating logical sentences with rules, as we did in the beginning of this chapter. To a datalog rule

$$\rho : R_1(u_1) \leftarrow R_2(u_2), \ldots, R_n(u_n)$$

we associate the logical sentence

$$\forall x_1, \ldots, x_m (R_1(u_1) \leftarrow R_2(u_2) \wedge \cdots \wedge R_n(u_n)),$$

where $x_1, \ldots, x_m$ are the variables occurring in the rule and $\leftarrow$ is the standard logical *implication*. Observe that an instance $\mathbf{I}$ satisfies ρ, denoted $\mathbf{I} \models \rho$, if for each instantiation

$$R_1(\nu(u_1)) \leftarrow R_2(\nu(u_2)), \ldots, R_n(\nu(u_n))$$

such that $R_2(\nu(u_2)), \ldots, R_n(\nu(u_n))$ belong to $\mathbf{I}$, so does $R_1(\nu(u_1))$. In the following, we do not distinguish between a rule ρ and the associated sentence. For a program P, the conjunction of the sentences associated with the rules of P is denoted by Σ_P.

It is useful to note that there are alternative ways to write the sentences associated with rules of programs. In particular, the formula

$$\forall x_1, \ldots, x_m (R_1(u_1) \leftarrow R_2(u_2) \wedge \cdots \wedge R_n(u_n))$$

is equivalent to

$$\forall x_1, \ldots, x_q (\exists x_{q+1}, \ldots, x_m (R_2(u_2) \wedge \cdots \wedge R_n(u_n)) \rightarrow R_1(u_1)),$$

where $x_1, \ldots, x_q$ are the variables occurring in the head. It is also logically equivalent to

$$\forall x_1, \ldots, x_m (R_1(u_1) \vee \neg R_2(u_2) \vee \cdots \vee \neg R_n(u_n)).$$

This last form is particularly interesting. Formulas consisting of a disjunction of literals of which at most one is positive are called in logic *Horn clauses*. A datalog program can thus be viewed as a set of (particular) Horn clauses.

We next discuss the issue of choosing, among the models of Σ_P, the particular model that is intended as the answer. This is not a hard problem for datalog, although (as we shall see in Chapter 15) it becomes much more involved if datalog is extended with negation. For datalog, the idea for choosing the intended model is simply that the model should not contain more facts than necessary for satisfying Σ_P. So the intended model is minimal in some natural sense. This is formalized next.

DEFINITION 12.2.1 Let P be a datalog program and $\mathbf{I}$ an instance over $edb(P)$. A *model* of P is an instance over $sch(P)$ satisfying Σ_P. The *semantics* of P on input $\mathbf{I}$, denoted $P(\mathbf{I})$, is the minimum model of P containing $\mathbf{I}$, if it exists.

Ans_1

Station
Odeon
St.-Michel
Chatelet
Louvres
Palais-Royal
Tuileries
Concorde

Ans_2

Line
4
1

Ans_3

⟨ ⟩

Figure 12.2: Relations of $P_{metro}(\mathbf{I})$

For P_{metro} as in Example 12.1.3, and $\mathbf{I}$ as in Fig. 12.1, the values of *Ans*_1, *Ans*_2, and *Ans*_3 in $P(\mathbf{I})$ are shown in Fig. 12.2.

We briefly discuss the choice of the minimal model at the end of this section.

Although the previous definition is natural, we cannot be entirely satisfied with it at this point:

- For given P and $\mathbf{I}$, we do not know (yet) whether the semantics of P is defined (i.e., whether there exists a minimum model of Σ_P containing $\mathbf{I}$).
- Even if such a model exists, the definition does not provide any algorithm for computing $P(\mathbf{I})$. Indeed, it is not (yet) clear that such an algorithm exists.

We next provide simple answers to both of these problems.

Observe that by definition, $P(\mathbf{I})$ is an instance over $sch(P)$. A priori, we must consider all instances over $sch(P)$, an infinite set. It turns out that it suffices to consider only those instances with active domain in $adom(P, \mathbf{I})$ (i.e., a finite set of instances). For given P and $\mathbf{I}$, let $\mathbf{B}(P, \mathbf{I})$ be the instance over $sch(P)$ defined by

1. For each R in $edb(P)$, a fact $R(u)$ is in $\mathbf{B}(P, \mathbf{I})$ iff it is in $\mathbf{I}$; and
2. For each R in $idb(P)$, each fact $R(u)$ with constants in $adom(P, \mathbf{I})$ is in $\mathbf{B}(P, \mathbf{I})$.

We now verify that $\mathbf{B}(P, \mathbf{I})$ is a model of P containing $\mathbf{I}$.

LEMMA 12.2.2 Let P be a datalog program and $\mathbf{I}$ an instance over $edb(P)$. Then $\mathbf{B}(P, \mathbf{I})$ is a model of P containing $\mathbf{I}$.

Proof Let $A_1 \leftarrow A_2, \ldots, A_n$ be an instantiation of some rule r in P such that $A_2, \ldots, A_n$ hold in $\mathbf{B}(P, \mathbf{I})$. Then consider A_1. Because each variable occurring in the head of r also occurs in the body, each constant occurring in A_1 belongs to $adom(P, \mathbf{I})$. Thus by definition 2 just given, A_1 is in $\mathbf{B}(P, \mathbf{I})$. Hence $\mathbf{B}(P, \mathbf{I})$ satisfies the sentence associated with that particular rule, so $\mathbf{B}(P, \mathbf{I})$ satisfies Σ_P. Clearly, $\mathbf{B}(P, \mathbf{I})$ contains $\mathbf{I}$ by definition 1. ■

Thus the semantics of P on input $\mathbf{I}$, if defined, is a subset of $\mathbf{B}(P, \mathbf{I})$. This means that there is no need to consider instances with constants outside $adom(P, \mathbf{I})$.

We next demonstrate that $P(\mathbf{I})$ is always defined.

THEOREM 12.2.3 Let P be a datalog program, $\mathbf{I}$ an instance over $edb(P)$, and $\mathcal{X}$ the set of models of P containing $\mathbf{I}$. Then

1. $\cap\mathcal{X}$ is the minimal model of P containing $\mathbf{I}$, so $P(\mathbf{I})$ is defined.
2. $adom(P(\mathbf{I})) \subseteq adom(P, \mathbf{I})$.
3. For each R in $edb(P)$, $P(\mathbf{I})(R) = \mathbf{I}(R)$.

Proof Note that $\mathcal{X}$ is nonempty, because $\mathbf{B}(P, \mathbf{I})$ is in $\mathcal{X}$. Let $r \equiv A_1 \leftarrow A_2, \ldots, A_n$ be a rule in P and ν a valuation of the variables occurring in the rule. To prove (1), we show that

(*) if $\nu(A_2), \ldots, \nu(A_n)$ are in $\cap\mathcal{X}$ then $\nu(A_1)$ is also in $\cap\mathcal{X}$.

For suppose that (*) holds. Then $\cap\mathcal{X} \models r$, so $\cap\mathcal{X}$ satisfies Σ_P. Because each instance in $\mathcal{X}$ contains $\mathbf{I}$, $\cap\mathcal{X}$ contains $\mathbf{I}$. Hence $\cap\mathcal{X}$ is a model of P containing $\mathbf{I}$. By construction, $\cap\mathcal{X}$ is minimal, so (1) holds.

To show (*), suppose that $\nu(A_2), \ldots, \nu(A_n)$ are in $\cap\mathcal{X}$ and let $\mathbf{K}$ be in $\mathcal{X}$. Because $\cap\mathcal{X} \subseteq \mathbf{K}$, $\nu(A_2), \ldots, \nu(A_n)$ are in $\mathbf{K}$. Because $\mathbf{K}$ is in $\mathcal{X}$, $\mathbf{K}$ is a model of P, so $\nu(A_1)$ is in $\mathbf{K}$. This is true for each $\mathbf{K}$ in $\mathcal{X}$. Hence $\nu(A_1)$ is in $\cap\mathcal{X}$ and (*) holds, which in turn proves (1).

By Lemma 12.2.2, $\mathbf{B}(P, \mathbf{I})$ is a model of P containing $\mathbf{I}$. Therefore $P(\mathbf{I}) \subseteq \mathbf{B}(P, \mathbf{I})$. Hence

- $adom(P(\mathbf{I})) \subseteq adom(\mathbf{B}(P, \mathbf{I})) = adom(P, \mathbf{I})$, so (2) holds.
- For each R in $edb(P)$, $\mathbf{I}(R) \subseteq P(\mathbf{I})(R)$ [because $P(\mathbf{I})$ contains $\mathbf{I}$] and $P(\mathbf{I})(R) \subseteq \mathbf{B}(P, \mathbf{I})(R) = \mathbf{I}(R)$; which shows (3). ■

The previous development also provides an algorithm for computing the semantics of datalog programs. Given P and $\mathbf{I}$, it suffices to consider all instances that are subsets of $\mathbf{B}(P, \mathbf{I})$, find those that are models of P and contain $\mathbf{I}$, and compute their intersection. However, this is clearly an inefficient procedure. The next section provides a more reasonable algorithm.

We conclude this section with two remarks on the definition of semantics of datalog programs. The first explains the choice of a minimal model. The second rephrases our definition in more standard logic-programming terminology.

Why Choose the Minimal Model?

This choice is the natural consequence of an implicit hypothesis of a philosophical nature: the *closed world assumption* (CWA) (see Chapter 2).

The CWA concerns the connection between the database and the world it models.

Clearly, databases are often incomplete (i.e., facts that may be true in the world are not necessarily recorded in the database). Thus, although we can reasonably assume that a fact recorded in the database is true in the world, it is not clear what we can say about facts not explicitly recorded. Should they be considered false, true, or unknown? The CWA provides the simplest solution to this problem: Treat the database as if it records complete information about the world (i.e., assume that all facts not in the database are false). This is equivalent to taking as true only the facts that must be true in all worlds modeled by the database. By extension, this justifies the choice of minimal model as the semantics of a datalog program. Indeed, the minimal model consists of the facts we know must be true in all worlds satisfying the sentences (and including the input instance). As we shall see, this has an equivalent proof-theoretic counterpart, which will justify the proof-theoretic semantics of datalog programs: Take as true precisely the facts that can be proven true from the input and the sentences corresponding to the datalog program. Facts that cannot be proven are therefore considered false.

Importantly, the CWA is not so simple to use in the presence of negation or disjunction. For example, suppose that a database holds $\{p \vee q\}$. Under the CWA, then both $\neg p$ and $\neg q$ are inferred. But the union $\{p \vee q, \neg p, \neg q\}$ is inconsistent, which is certainly not the intended result.

Herbrand Interpretation

We relate briefly the semantics given to datalog programs to standard logic-programming terminology.

In logic programming, the facts of an input instance $\mathbf{I}$ are not separated from the sentences of a datalog program P. Instead, sentences stating that all facts in $\mathbf{I}$ are true are included in P. This gives rise to a logical theory $\Sigma_{P,\mathbf{I}}$ consisting of the sentences in Σ_P and of one sentence $P(u)$ [sometimes written $P(u) \leftarrow$] for each fact $P(u)$ in the instance. The semantics is defined as a particular model of this set of sentences. A problem is that standard interpretations in first-order logic permit interpretation of constants of the theory with arbitrary elements of the domain. For instance, the constants *Odeon* and *St.-Michel* may be interpreted by the same element (e.g., *John*). This is clearly not what we mean in the database context. We wish to interpret *Odeon* by *Odeon* and similarly for all other constants. Interpretations that use the identity function to interpret the constant symbols are called *Herbrand interpretations* (see Chapter 2). (If function symbols are present, restrictions are also placed on how terms involving functions are interpreted.) Given a set Γ of formulas, a *Herbrand model* of Γ is a Herbrand interpretation satisfying Γ.

Thus in logic programming terms, the semantics of a program P given an instance $\mathbf{I}$ can be viewed as the minimum Herbrand model of $\Sigma_{P,\mathbf{I}}$.

12.3 Fixpoint Semantics

In this section, we present an operational semantics for datalog programs stemming from fixpoint theory. We use an operator called the *immediate consequence* operator. The operator produces new facts starting from known facts. We show that the model-theoretic se-

mantics, $P(\mathbf{I})$, can also be defined as the smallest solution of a fixpoint equation involving that operator. It turns out that this solution can be obtained constructively. This approach therefore provides an alternative constructive definition of the semantics of datalog programs. It can be viewed as an implementation of the model-theoretic semantics.

Let P be a datalog program and $\mathbf{K}$ an instance over $sch(P)$. A fact A is an *immediate consequence* for $\mathbf{K}$ and P if either $A \in \mathbf{K}(R)$ for some *edb* relation R, or $A \leftarrow A_1, \ldots, A_n$ is an instantiation of a rule in P and each A_i is in $\mathbf{K}$. The *immediate consequence operator* of P, denoted T_P, is the mapping from $inst(sch(P))$ to $inst(sch(P))$ defined as follows. For each $\mathbf{K}$, $T_P(\mathbf{K})$ consists of all facts A that are immediate consequences for $\mathbf{K}$ and P.

We next note some simple mathematical properties of the operator T_P over sets of instances. We first define two useful properties. For an operator T,

- T is *monotone* if for each $\mathbf{I}, \mathbf{J}$, $\mathbf{I} \subseteq \mathbf{J}$ implies $T(\mathbf{I}) \subseteq T(\mathbf{J})$.
- $\mathbf{K}$ is a *fixpoint* of T if $T(\mathbf{K}) = \mathbf{K}$.

The proof of the next lemma is straightforward and is omitted (see Exercise 12.9).

Lemma 12.3.1 Let P be a datalog program.

(i) The operator T_P is monotone.

(ii) An instance $\mathbf{K}$ over $sch(P)$ is a model of Σ_P iff $T_P(\mathbf{K}) \subseteq \mathbf{K}$.

(iii) Each fixpoint of T_P is a model of Σ_P; the converse does not necessarily hold.

It turns out that $P(\mathbf{I})$ (as defined by the model-theoretic semantics) is a fixpoint of T_P. In particular, it is the minimum fixpoint containing $\mathbf{I}$. This is shown next.

Theorem 12.3.2 For each P and $\mathbf{I}$, T_P has a minimum fixpoint containing $\mathbf{I}$, which equals $P(\mathbf{I})$.

Proof Observe first that $P(\mathbf{I})$ is a fixpoint of T_P:

- $T_P(P(\mathbf{I})) \subseteq P(\mathbf{I})$ because $P(\mathbf{I})$ is a model of P; and
- $P(\mathbf{I}) \subseteq T_P(P(\mathbf{I}))$. [Because $T_P(P(\mathbf{I})) \subseteq P(\mathbf{I})$ and T_P is monotone, $T_P(T_P(P(\mathbf{I})))$ $\subseteq T_P(P(\mathbf{I}))$. Thus $T_P(P(\mathbf{I}))$ is a model of Σ_P. Because T_P preserves the contents of the *edb* relations and $\mathbf{I} \subseteq P(\mathbf{I})$, we have $\mathbf{I} \subseteq T_P(P(\mathbf{I}))$. Thus $T_P(P(\mathbf{I}))$ is a model of Σ_P containing $\mathbf{I}$. Because $P(\mathbf{I})$ is the minimum such model, $P(\mathbf{I}) \subseteq T_P(P(\mathbf{I}))$.]

In addition, each fixpoint of T_P containing $\mathbf{I}$ is a model of P and thus contains $P(\mathbf{I})$ (which is the intersection of all models of P containing $\mathbf{I}$). Thus $P(\mathbf{I})$ is the minimum fixpoint of P containing $\mathbf{I}$. ■

The fixpoint definition of the semantics of P presents the advantage of leading to a constructive definition of $P(\mathbf{I})$. In logic programming, this is shown using fixpoint theory (i.e., using Knaster-Tarski's and Kleene's theorems). However, the database framework is much simpler than the general logic-programming one, primarily due to the lack of function symbols. We therefore choose to show the construction directly, without the formidable machinery of the theory of fixpoints in complete lattices. In Remark 12.3.5

we sketch the more standard proof that has the advantage of being applicable to the larger context of logic programming.

Given an instance $\mathbf{I}$ over $edb(P)$, one can compute $T_P(\mathbf{I})$, $T_P^2(\mathbf{I})$, $T_P^3(\mathbf{I})$, etc. Clearly,

$$\mathbf{I} \subseteq T_P(\mathbf{I}) \subseteq T_P^2(\mathbf{I}) \subseteq T_P^3(\mathbf{I}) \ldots \subseteq \mathbf{B}(P, \mathbf{I}).$$

This follows immediately from the fact that $\mathbf{I} \subseteq T_P(\mathbf{I})$ and the monotonicity of T_P. Let N be the number of facts in $\mathbf{B}(P, \mathbf{I})$. (Observe that N depends on $\mathbf{I}$.) The sequence $\{T_P^i(\mathbf{I})\}_i$ reaches a fixpoint after at most N steps. That is, for each $i \geq N$, $T_P^i(\mathbf{I}) = T_P^N(\mathbf{I})$. In particular, $T_P(T_P^N(\mathbf{I})) = T_P^N(\mathbf{I})$, so $T_P^N(\mathbf{I})$ is a fixpoint of T_P. We denote this fixpoint by $T_P^\omega(\mathbf{I})$.

EXAMPLE 12.3.3 Recall the program P_{TC} for computing the transitive closure of a graph G:

$$T(x, y) \leftarrow G(x, y)$$
$$T(x, y) \leftarrow G(x, z), T(z, y).$$

Consider the input instance

$$\mathbf{I} = \{G(1, 2), G(2, 3), G(3, 4), G(4, 5)\}.$$

Then we have

$$T_{P_{TC}}(I) = I \cup \{T(1, 2), T(2, 3), T(3, 4), T(4, 5)\}$$
$$T_{P_{TC}}^2(I) = T_{P_{TC}}(I) \cup \{T(1, 3), T(2, 4), T(3, 5)\}$$
$$T_{P_{TC}}^3(I) = T_{P_{TC}}^2(I) \cup \{T(1, 4), T(2, 5)\}$$
$$T_{P_{TC}}^4(I) = T_{P_{TC}}^3(I) \cup \{T(1, 5)\}$$
$$T_{P_{TC}}^5(I) = T_{P_{TC}}^4(I).$$

Thus $T_{P_{TC}}^\omega(I) = T_{P_{TC}}^4(I)$.

We next show that $T_P^\omega(\mathbf{I})$ is exactly $P(\mathbf{I})$ for each datalog program P.

THEOREM 12.3.4 Let P be a datalog program and $\mathbf{I}$ an instance over $edb(P)$. Then $T_P^\omega(\mathbf{I}) = P(\mathbf{I})$.

Proof By Theorem 12.3.2, it suffices to show that $T_P^\omega(\mathbf{I})$ is the minimum fixpoint of T_P containing $\mathbf{I}$. As noted earlier,

$$T_P(T_P^\omega(\mathbf{I})) = T_P(T_P^N(\mathbf{I})) = T_P^N(\mathbf{I}) = T_P^\omega(\mathbf{I}).$$

where N is the number of facts in $\mathbf{B}(P, \mathbf{I})$. Therefore $T_P^\omega(\mathbf{I})$ is a fixpoint of T_P that contains $\mathbf{I}$.

To show that it is minimal, consider an arbitrary fixpoint $\mathbf{J}$ of T_P containing $\mathbf{I}$. Then $\mathbf{J} \supseteq T_P^0(\mathbf{I}) = \mathbf{I}$. By induction on i, $\mathbf{J} \supseteq T_P^i(\mathbf{I})$ for each i, so $\mathbf{J} \supseteq T_P^\omega(\mathbf{I})$. Thus $T_P^\omega(\mathbf{I})$ is the minimum fixpoint of T_P containing $\mathbf{I}$. ■

The smallest integer i such that $T_P^i(\mathbf{I}) = T_P^\omega(\mathbf{I})$ is called the stage for P and $\mathbf{I}$ and is denoted $stage(P, \mathbf{I})$. As already noted, $stage(P, \mathbf{I}) \leq N = |\mathbf{B}(P, \mathbf{I})|$.

Evaluation

The fixpoint approach suggests a straightforward algorithm for the evaluation of datalog. We explain the algorithm in an example. We extend relational algebra with a *while* operator that allows us to iterate an algebraic expression while some condition holds. (The resulting language is studied extensively in Chapter 17.)

Consider again the transitive closure query. We wish to compute the transitive closure of relation G in relation T. Both relations are over AB. This computation is performed by the following program:

$$T := G;$$
$$\text{while } q(T) \neq T \text{ do } T := q(T);$$

where

$$q(T) = G \cup \pi_{AB}(\delta_{B \to C}(G) \bowtie \delta_{A \to C}(T)).$$

(Recall that δ is the renaming operation as introduced in Chapter 4.)

Observe that q is an SPJRU expression. In fact, at each step, q computes the immediate consequence operator T_P, where P is the transitive closure datalog program in Example 12.3.3. One can show in general that the immediate consequence operator can be computed using SPJRU expressions (i.e., relational algebra without the difference operation). Furthermore, the SPJRU expressions extended carefully with a while construct yield exactly the expressive power of datalog. The test of the while is used to detect when the fixpoint is reached.

The while construct is needed only for recursion. Let us consider again the nonrecursive datalog of Chapter 4. Let P be a datalog program. Consider the graph $(sch(P), E_P)$, where $\langle S, S' \rangle$ is an edge in E_P if S' occurs in the head of some rule r in P and S occurs in the body of r. Then P is *nonrecursive* if the graph is acyclic. We mentioned already that nr-datalog programs are equivalent to SPJRU queries (see Section 4.5). It is also easy to see that, for each nr-datalog program P, there exists a constant d such that for each $\mathbf{I}$ over $edb(P)$, $stage(P, \mathbf{I}) \leq d$. In other words, the fixpoint is reached after a bounded number of steps, dependent only on the program. (See Exercise 12.29.) Programs for which this happens are called *bounded*. We examine this property in more detail in Section 12.5.

A lot of redundant computation is performed when running the preceding transitive closure program. We study optimization techniques for datalog evaluation in Chapter 13.

Remark 12.3.5 In this remark, we make a brief excursion into standard fixpoint theory to reprove Theorem 12.3.4. This machinery is needed when proving the analog of that theorem in the more general context of logic programming. A partially ordered set $(U, \leq)$ is a *complete lattice* if each subset has a least upper bound and a greatest lower bound, denoted *sup* and *inf*, respectively. In particular, $inf(U)$ is denoted $\bot$ and $sup(U)$ is denoted $\top$. An operator T on U is *monotone* iff for each $x, y \in U$, $x \leq y$ implies $T(x) \leq T(y)$. An operator T on U is *continuous* if for each subset V, $T(sup(V)) = sup(T(V))$. Note that continuity implies monotonicity.

To each datalog program P and instance $\mathbf{I}$, we associate the program $P_\mathbf{I}$ consisting of the rules of P and one rule $R(u) \leftarrow$ for each fact $R(u)$ in $\mathbf{I}$. We consider the complete lattice formed with $(inst(sch(P)), \subseteq)$ and the operator $T_{P_\mathbf{I}}$ defined by the following: For each $\mathbf{K}$, a fact A is in $T_{P_\mathbf{I}}(\mathbf{K})$ if A is an immediate consequence for $\mathbf{K}$ and $P_\mathbf{I}$. The operator $T_{P_\mathbf{I}}$ on $(inst(sch(P)), \subseteq)$ is continuous (so also monotone).

The Knaster-Tarski theorem states that a monotone operator in a complete lattice has a least fixpoint that equals $inf(\{x \mid x \in U, T(x) \leq x\})$. Thus the least fixpoint of $T_{P_\mathbf{I}}$ exists. Fixpoint theory also provides the constructive definition of the least fixpoint for continuous operators. Indeed, Kleene's theorem states that if T is a continuous operator on a complete lattice, then its least fixpoint is $sup(\{\mathbf{K}_i \mid i \geq 0\})$ where $\mathbf{K}_0 = \bot$ and for each $i > 0$, $\mathbf{K}_i = T(\mathbf{K}_{i-1})$. Now in our case, $\bot = \emptyset$ and

$$\emptyset \cup T_{P_\mathbf{I}}(\emptyset) \cup \cdots \cup T^i_{P_\mathbf{I}}(\emptyset) \cup \cdots$$

coincides with $P(\mathbf{I})$.

In logic programming, function symbols are also considered (see Example 12.1.4). In this context, the sequence of $\{T^i_{P_\mathbf{I}}(\mathbf{I})\}_{i>0}$ does not generally converge in a finite number of steps, so the fixpoint evaluation is no longer constructive. However, it does converge in countably many steps to the least fixpoint $\cup\{T^i_{P_\mathbf{I}}(\emptyset) \mid i \geq 0\}$. Thus fixpoint theory is useful primarily when dealing with logic programs with function symbols. It is an overkill in the simpler context of datalog. ■

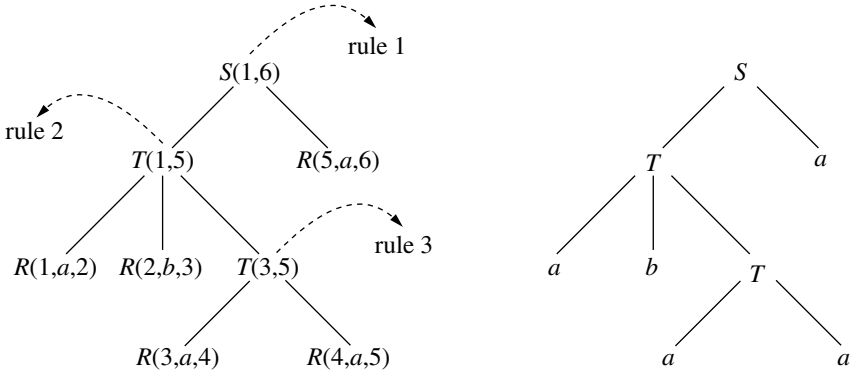
12.4 Proof-Theoretic Approach

Another way of defining the semantics of datalog is based on proofs. The basic idea is that the answer of a program P on $\mathbf{I}$ consists of the set of facts that can be proven using P and $\mathbf{I}$. The result turns out to coincide, again, with $P(\mathbf{I})$.

The first step is to define what is meant by *proof*. A *proof tree* of a fact A from $\mathbf{I}$ and P is a labeled tree where

1. each vertex of the tree is labeled by a fact;
2. each leaf is labeled by a fact in $\mathbf{I}$;
3. the root is labeled by A; and
4. for each internal vertex, there exists an instantiation $A_1 \leftarrow A_2, \ldots, A_n$ of a rule in P such that the vertex is labeled A_1 and its children are respectively labeled $A_2, \ldots, A_n$.

Such a tree provides a proof of the fact A.



(a) Datalog proof (b) Context-free derivation

Figure 12.3: Proof tree

Example 12.4.1 Consider the following program:

$$S(x_1, x_3) \leftarrow T(x_1, x_2), R(x_2, a, x_3)$$
$$T(x_1, x_4) \leftarrow R(x_1, a, x_2), R(x_2, b, x_3), T(x_3, x_4)$$
$$T(x_1, x_3) \leftarrow R(x_1, a, x_2), R(x_2, a, x_3)$$

and the instance

$$\{R(1, a, 2), R(2, b, 3), R(3, a, 4), R(4, a, 5), R(5, a, 6)\}.$$

A proof tree of $S(1, 6)$ is shown in Fig. 12.3(a).

The reader familiar with context-free languages will notice the similarity between proof trees and derivation trees in context-free languages. This connection is especially strong in the case of datalog programs that have the form of the one in Example 12.4.1. This will be exploited in the last section of this chapter.

Proof trees provide proofs of facts. It is straightforward to show that a fact A is in $P(\mathbf{I})$ iff there exists a proof tree for A from $\mathbf{I}$ and P. Now given a fact A to prove, one can look for a proof either *bottom up* or *top down*.

The bottom-up approach is an alternative way of looking at the constructive fixpoint technique. One begins with the facts from $\mathbf{I}$ and then uses the rules to infer new facts, much like the immediate consequence operator. This is done repeatedly until no new facts can be inferred. The rules are used as "factories" producing new facts from already proven ones. This eventually yields all facts that can be proven and is essentially the same as the fixpoint approach.

In contrast to the bottom-up and fixpoint approaches, the top-down approach allows one to direct the search for a proof when one is only interested in proving particular facts.

For example, suppose the query $Ans\_1(Louvre)$ is posed against the program P_{metro} of Example 12.1.3, with the input instance of Fig. 12.1. Then the top-down approach will never consider atoms involving stations on Line 9, intuitively because they are are not reachable from *Odeon* or *Louvre*. More generally, the top-down approach inhibits the indiscriminate inference of facts that are irrelevant to the facts of interest.

The top-down approach is described next. This takes us to the field of logic programming. But first we need some notation, which will remind us once again that "To bar an easy access to newcomers every scientific domain has introduced its own terminology and notation" [Apt91].

Notation

Although we already borrowed a lot of terminology and notation from the logic-programming field (e.g., term, fact, atom), we must briefly introduce some more.

A *positive literal* is an atom [i.e., $P(u)$ for some free tuple u]; and a *negative literal* is the negation of one [i.e., $\neg P(u)$]. A formula of the form

$$\forall x_1, \ldots, x_m (A_1 \vee \cdots \vee A_n \vee \neg B_1 \vee \cdots \vee \neg B_p),$$

where the A_i, B_j are positive literals, is called a *clause*. Such a clause is written in *clausal form* as

$$A_1, \ldots, A_n \leftarrow B_1, \ldots, B_p.$$

A clause with a single literal in the head ($n = 1$) is called a *definite clause*. A definite clause with an empty body is called a *unit clause*. A clause with no literal in the head is called a *goal clause*. A clause with an empty body and head is called an *empty clause* and is denoted $\square$. Examples of these and their logical counterparts are as follows:

definite	$T(x, y) \leftarrow R(x, z), T(z, y)$	$T(x, y) \vee \neg R(x, z) \vee \neg T(z, y)$
unit	$T(x, y) \leftarrow$	$T(x, y)$
goal	$\leftarrow R(x, z), T(z, y)$	$\neg R(x, z) \vee \neg T(z, y)$
empty	$\square$	*false*

The empty clause is interpreted as a contradiction. Intuitively, this is because it corresponds to the disjunction of an empty set of formulas.

A *ground* clause is a clause with no occurrence of variables.

The top-down proof technique introduced here is called SLD resolution. Goals serve as the basic focus of activity in SLD resolution. As we shall see, the procedure begins with a goal such as $\leftarrow St\_Reachable(x, Concorde)$, $Li\_Reachable(x, 9)$. A correct answer of this goal on input $\mathbf{I}$ is any value a such that $St\_Reachable(a, Concorde)$ and $Li\_Reachable(a, 9)$ are implied by $\Sigma_{P_{metro}, \mathbf{I}}$. Furthermore, each intermediate step of the top-down approach consists of obtaining a new goal from a previous goal. Finally, the procedure is deemed successful if the final goal reached is empty.

The standard exposition of SLD resolution is based on definite clauses. There is a

subtle distinction between datalog rules and definite clauses: For datalog rules, we imposed the restriction that each variable that occurs in the head also appears in the body. (In particular, a datalog unit clause must be ground.) We will briefly mention some minor consequences of this distinction.

As already introduced in Remark 12.3.5, to each datalog program P and instance $\mathbf{I}$, we associate the program $P_{\mathbf{I}}$ consisting of the rules of P and one rule $R(u) \leftarrow$ for each fact $R(u)$ in $\mathbf{I}$. Therefore in the following we ignore the instance $\mathbf{I}$ and focus on programs that already integrate all the known facts in the set of rules. We denote such a program $P_{\mathbf{I}}$ to emphasize its relationship to an instance $\mathbf{I}$. Observe that from a semantic point of view

$$P(\mathbf{I}) = P_{\mathbf{I}}(\emptyset).$$

This ignores the distinction between *edb* and *idb* relations, which no longer exists for $P_{\mathbf{I}}$.

Example 12.4.2 Consider the program P and instance $\mathbf{I}$ of Example 12.4.1. The rules of $P_{\mathbf{I}}$ are

1. $S(x_1, x_3) \leftarrow T(x_1, x_2), R(x_2, a, x_3)$
2. $T(x_1, x_4) \leftarrow R(x_1, a, x_2), R(x_2, b, x_3), T(x_3, x_4)$
3. $T(x_1, x_3) \leftarrow R(x_1, a, x_2), R(x_2, a, x_3)$
4. $R(1, a, 2) \leftarrow$
5. $R(2, b, 3) \leftarrow$
6. $R(3, a, 4) \leftarrow$
7. $R(4, a, 5) \leftarrow$
8. $R(5, a, 6) \leftarrow$

Warm-Up

Before discussing SLD resolution, as a warm-up we look at a simplified version of the technique by considering only ground rules. To this end, consider a datalog program $P_{\mathbf{I}}$ (integrating the facts) consisting only of fully instantiated rules (i.e., with no occurrences of variables). Consider a ground goal $g \equiv$

$$\leftarrow A_1, \ldots, A_i, \ldots, A_n$$

and some (ground) rule $r \equiv A_i \leftarrow B_1, \ldots, B_m$ in $P_{\mathbf{I}}$. A *resolvent* of g with r is the ground goal

$$\leftarrow A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n.$$

Viewed as logical sentences, the resolvent of g with r is actually implied by g and r. This is best seen by writing these explicitly as clauses:

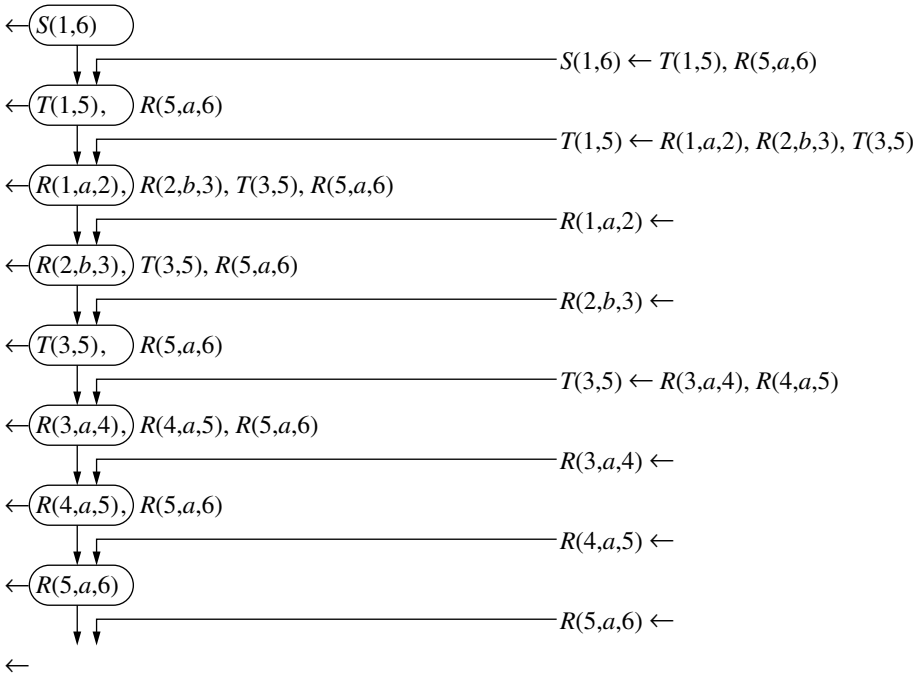


Figure 12.4: SLD ground refutation

$$(\neg A_1 \vee \cdots \vee \neg A_i \vee \cdots \vee \neg A_n) \wedge (A_i \vee \neg B_1 \vee \cdots \vee \neg B_m)$$
$$\Rightarrow (\neg A_1 \vee \cdots \vee \neg A_{i-1} \vee \neg B_1 \vee \cdots \vee \neg B_m \vee \neg A_{i+1} \vee \cdots \vee \neg A_n).$$

In general, the converse does not hold.

A *derivation* from g with $P_{\mathbf{I}}$ is a sequence of goals $g \equiv g_0, g_1, \ldots$ such that for each $i > 0$, g_i is a resolvent of g_{i-1} with some rule in $P_{\mathbf{I}}$. We will see that to prove a fact A, it suffices to exhibit a *refutation* of $\leftarrow A$—that is, a derivation

$$g_0 \equiv \leftarrow A, \quad g_1, \ldots, g_i, \ldots, \quad g_q \equiv \square.$$

Example 12.4.3 Consider Example 12.4.1 and the program obtained by all possible instantiations of the rules of $P_{\mathbf{I}}$ in Example 12.4.2. An SLD ground refutation is shown in Fig. 12.4. It is a refutation of $\leftarrow S(1, 6)$ [i.e. a proof of $S(1, 6)$].

Let us now explain why refutations provide proofs of facts. Suppose that we wish to prove $A_1 \wedge \cdots \wedge A_n$. To do this we may equivalently prove that its negation (i.e. $\neg A_1 \vee \cdots \vee \neg A_n$) is false. In other words, we try to refute (or disprove) $\leftarrow A_1, \ldots, A_n$. The following rephrasing of the refutation in Fig. 12.4 should make this crystal clear.

Example 12.4.4 Continuing with the previous example, to prove $S(1, 6)$, we try to refute its negation [i.e., $\neg S(1, 6)$ or $\leftarrow S(1, 6)$]. This leads us to considering, in turn, the formulas

Goal	Rule used
$\neg S(1, 6)$	(1)
$\Rightarrow \neg T(1, 5) \vee \neg R(5, a, 6)$	(2)
$\Rightarrow \neg R(1, a, 2) \vee \neg R(2, b, 3) \vee \neg T(3, 5) \vee \neg R(5, a, 6)$	(4)
$\Rightarrow \neg R(2, b, 3) \vee \neg T(3, 5) \vee \neg R(5, a, 6)$	(5)
$\Rightarrow \neg T(3, 5) \vee \neg R(5, a, 6)$	(3)
$\Rightarrow \neg R(3, a, 4) \vee \neg R(4, a, 5) \vee \neg R(5, a, 6)$	(6)
$\Rightarrow \neg R(4, a, 5) \vee \neg R(5, a, 6)$	(7)
$\Rightarrow \neg R(5, a, 6)$	(8)
$\Rightarrow$ *false*	

At the end of the derivation, we have obtained a contradiction. Thus we have *refuted* $\neg S(1, 6)$ [i.e., proved $S(1, 6)$].

Thus refutations provide proofs. As a consequence, a goal can be thought of as a query. Indeed, the arrow is sometimes denoted with a question mark in goals. For instance, we sometimes write

$$\text{?- } S(1, 6) \text{ for } \leftarrow S(1, 6).$$

Observe that the process of finding a proof is nondeterministic for two reasons: the choice of the literal A to replace and the rule that is used to replace it.

We now have a technique for proving facts. The benefit of this technique is that it is sound and complete, in the sense that the set of facts in $P(\mathbf{I})$ coincides with the facts that can be proven from $P_{\mathbf{I}}$.

Theorem 12.4.5 Let $P_{\mathbf{I}}$ be a datalog program and $ground(P_{\mathbf{I}})$ be the set of instantiations of rules in $P_{\mathbf{I}}$ with values in $adom(P, \mathbf{I})$. Then for each ground goal g, $P_{\mathbf{I}}(\emptyset) \models \neg g$ iff there exists a refutation of g with $ground(P_{\mathbf{I}})$.

Crux To show the "only if," we prove by induction that

(**) for each ground goal g, if $T^i_{P_{\mathbf{I}}}(\emptyset) \models \neg g$, there exists a refutation of g with $ground(P_{\mathbf{I}})$.

(The "if" part is proved similarly by induction on the length of the refutation. Its proof is left for Exercise 12.18.)

The base case is obvious. Now suppose that (**) holds for some $i \geq 0$, and let $A_1, \ldots, A_m$ be ground atoms such that $T^{i+1}_{P_{\mathbf{I}}}(\emptyset) \models A_1 \wedge \cdots \wedge A_m$. Therefore each A_j is in $T^{i+1}_{P_{\mathbf{I}}}(\emptyset)$. Consider some j. If A_j is an *edb* fact, we are back to the base case. Otherwise

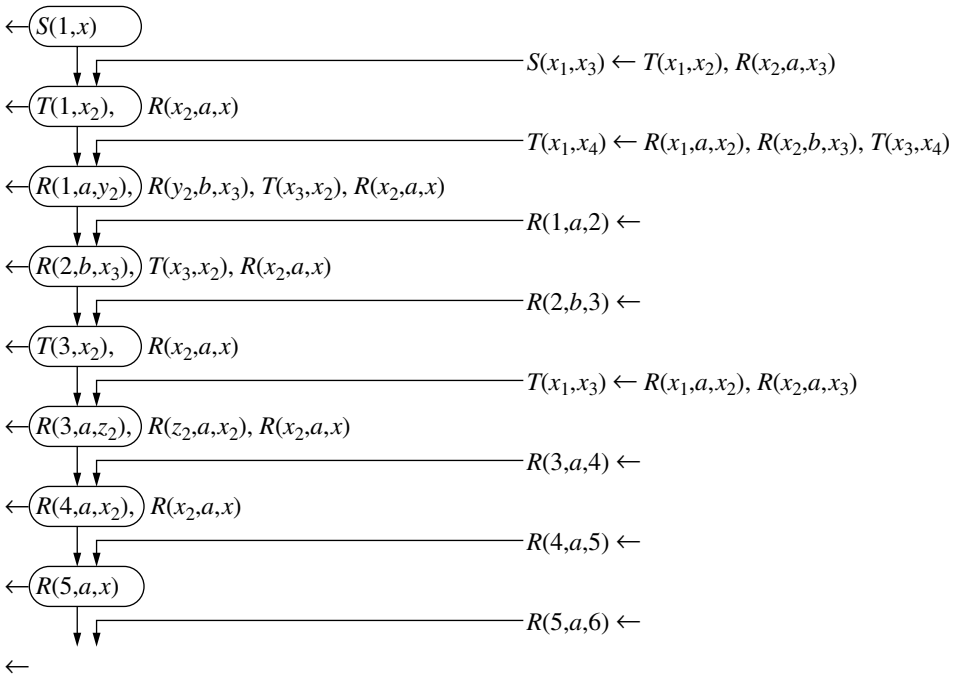


Figure 12.5: SLD refutation

there exists an instantiation $A_j \leftarrow B_1, \ldots, B_p$ of some rule in $P_{\mathbf{I}}$ such that $B_1, \ldots, B_p$ are in $T^i_{P_{\mathbf{I}}}(\emptyset)$. The refutation of $\leftarrow A_j$ with $ground(P_{\mathbf{I}})$ is as follows. It starts with

$$\leftarrow A_j$$
$$\leftarrow B_1, B_2 \ldots, B_p.$$

Now by induction there exist refutations of $\leftarrow B_n, 1 \leq n \leq p$, with $ground(P_{\mathbf{I}})$. Using these refutations, one can extend the preceding derivation to a derivation leading to the empty clause. Furthermore, the refutations for each of the A_j's can be combined to obtain a refutation of $\leftarrow A_1, \ldots, A_m$ as desired. Therefore (**) holds for $i + 1$. By induction, (**) holds. ■

SLD Resolution

The main difference between the general case and the warm-up is that we now handle goals and tuples with variables rather than just ground ones. In addition to obtaining the goal $\square$, the process determines an instantiation θ for the free variables of the goal g, such that $P_{\mathbf{I}}(\emptyset) \models \neg\theta g$. We start with an example: An SLD refutation of $\leftarrow S(1, x)$ is shown in Fig. 12.5.

In general, we start with a goal (which does not have to be ground):

$$\leftarrow A_1, \ldots, A_i, \ldots, A_n.$$

Suppose that we selected a literal to be replaced [e.g., $A_i = Q(1, x_2, x_5)$]. Any rule used for the replacement must have Q for predicate in the head, just as in the ground case. For instance, we might try some rule

$$Q(x_1, x_4, x_3) \leftarrow P(x_1, x_2),\ P(x_2, x_3),\ Q(x_3, x_4, x_5).$$

We now have two difficulties:

(i) The same variable may occur in the selected literal and in the rule with two different meanings. For instance, x_2 in the selected literal is not to be confused with x_2 in the rule.

(ii) The pattern of constants and of equalities between variables in the selected literal and in the head of the rule may be different. In our example, for the first attribute we have 1 in the selected literal and a variable in the rule head.

The first of these two difficulties is handled easily by renaming the variables of the rules. We shall use the following renaming discipline: Each time a rule is used, a new set of distinct variables is substituted for the ones in the rule. Thus we might use instead the rule

$$Q(x_{11}, x_{14}, x_{13}) \leftarrow P(x_{11}, x_{12}),\ P(x_{12}, x_{13}),\ Q(x_{13}, x_{14}, x_{15}).$$

The second difficulty requires a more careful approach. It is tackled using unification, which matches the pattern of the selected literal to that of the head of the rule, if possible. In the example, unification consists of finding a substitution θ such that $\theta(Q(1, x_2, x_5)) = \theta(Q(x_{11}, x_{14}, x_{13}))$. Such a substitution is called a unifier. For example, the substitution $\theta(x_{11}) = 1,\ \theta(x_2) = \theta(x_{14}) = \theta(x_5) = \theta(x_{13}) = y$ is a unifier for $Q(1, x_2, x_5)$ and $Q(x_{11}, x_{14}, x_{13})$, because $\theta(Q(1, x_2, x_5)) = \theta(Q(x_{11}, x_{14}, x_{13})) = Q(1, y, y)$. Note that this particular unifier is unnecessarily restrictive; there is no reason to identify all of x_2, x_3, x_4, x_5.

A unifier that is no more restrictive than needed to unify the atoms is called a most general unifier (mgu). Applying the mgu to the rule to be used results in specializing the rule just enough so that it applies to the selected literal. These terms are formalized next.

Definition 12.4.6 Let A, B be two atoms. A *unifier* for A and B is a substitution θ such that $\theta A = \theta B$. A substitution θ is *more general* than a substitution ν, denoted $\theta \hookrightarrow \nu$, if for some substitution ν', $\nu = \theta \circ \nu'$. A *most general unifier* (mgu) for A and B is a unifier θ for A, B such that, for each unifier ν of A, B, we have $\theta \hookrightarrow \nu$.

Clearly, the relation $\hookrightarrow$ between unifiers is reflexive and transitive but not antisymmetric. Let $\approx$ be the equivalence relation on substitutions defined by $\theta \approx \nu$ iff $\theta \hookrightarrow \nu$ and $\nu \hookrightarrow \theta$. If $\theta \approx \nu$, then for each atom A, $\theta(A)$ and $\nu(A)$ are the same modulo renaming of variables.

Computing the mgu

We now develop an algorithm for computing an mgu for two atoms. Let R be a relation of arity p and $R(x_1, \ldots, x_p)$, $R(y_1, \ldots, y_p)$ two literals with disjoint sets of variables. Compute $\equiv$, the equivalence relation on **var** $\cup$ **dom** defined as the reflexive, transitive closure of: $x_i \equiv y_i$ for each i in $[1, p]$. The mgu of $R(x_1, \ldots, x_p)$ and $R(y_1, \ldots, y_p)$ does not exist if two distinct constants are in the same equivalence class. Otherwise their mgu is the substitution θ such that

1. If $z \equiv a$ for some constant a, $\theta(z) = a$;
2. Otherwise $\theta(z) = z'$, where z' is the smallest (under a fixed ordering on **var**) such that $z \equiv z'$.

We show that the foregoing computes an mgu.

Lemma 12.4.7 The substitution θ just computed is an mgu for $R(x_1, \ldots, x_p)$ and $R(y_1, \ldots, y_p)$.

Proof Clearly, θ is a unifier for $R(x_1, \ldots, x_p)$ and $R(y_1, \ldots, y_p)$. Suppose ν is another unifier for the same atoms. Let $\equiv_\nu$ be the equivalence relation on **var** $\cup$ **dom** defined by $x \equiv_\nu y$ iff $\nu(x) = \nu(y)$. Because ν is a unifier, $\nu(x_i) = \nu(y_i)$. It follows that $x_i \equiv_\nu y_i$, so $\equiv$ refines $\equiv_\nu$. Then the substitution ν' defined by $\nu'(\theta(x)) = \nu(x)$, is well defined, because $\theta(x) = \theta(x')$ implies $\nu(x) = \nu(x')$. Thus $\nu = \theta \circ \nu'$ so $\theta \hookrightarrow \nu$. Because this holds for every unifier ν, it follows that θ is an mgu for the aforementioned atoms. ■

The following facts about mgu's are important to note. Their proof is left to the reader (Exercise 12.19). In particular, part (ii) of the lemma says that the mgu of two atoms, if it exists, is essentially unique (modulo renaming of variables).

Lemma 12.4.8 Let A, B be atoms.

(i) If there exists a unifier for A, B, then A, B have an mgu.

(ii) If θ and θ' are mgu's for A, B then $\theta \approx \theta'$.

(iii) Let A, B be atoms with mgu θ. Then for each atom C, if $C = \theta_1 A = \theta_2 B$ for substitutions θ_1, θ_2, then $C = \theta_3(\theta(A)) = \theta_3(\theta(B))$ for some substitution θ_3.

We are now ready to rephrase the notion of *resolvent* to incorporate variables. Let

$$g \equiv \leftarrow A_1, \ldots, A_i, \ldots, A_n, \qquad r \equiv B_1 \leftarrow B_2, \ldots, B_m$$

be a goal and a rule such that

1. g and r have no variable in common (which can always be ensured by renaming the variables of the rule).
2. A_i and B_1 have an mgu θ.

Then the *resolvent* of g with r using θ is the goal

$$\leftarrow \theta(A_1), \ldots, \theta(A_{i-1}), \theta(B_2), \ldots, \theta(B_m), \theta(A_{i+1}), \ldots, \theta(A_n).$$

As before, it is easily verified that this resolvent is implied by g and r.

An *SLD derivation* from a goal g with a program $P_{\mathbf{I}}$ is a sequence $g_0 = g, g_1, \ldots$ of goals and $\theta_0, \ldots$ of substitutions such that for each j, g_j is the resolvent of g_{j-1} with some rule in $P_{\mathbf{I}}$ using θ_{j_1}. An *SLD refutation* of a goal g with $P_{\mathbf{I}}$ is an SLD derivation $g_0 = g, \ldots, g_q = \square$ with $P_{\mathbf{I}}$.

We now explain the meaning of such a refutation. As in the variable-free case, the existence of a refutation of a goal $\leftarrow A_1, \ldots, A_n$ with $P_{\mathbf{I}}$ can be viewed as a proof of the negation of the goal. The goal is

$$\forall x_1, \ldots, x_m(\neg A_1 \vee \cdots \vee \neg A_n)$$

where $x_1, \ldots, x_m$ are the variables in the goal. Its negation is therefore equivalent to

$$\exists x_1, \ldots, x_m(A_1 \wedge \cdots \wedge A_n),$$

and the refutation can be seen as a proof of its validity. Note that, in the case of datalog programs (where by definition all unit clauses are ground), the composition $\theta_1 \circ \cdots \circ \theta_q$ of mgu's used while refuting the goal yields a substitution by constants. This substitution provides "witnesses" for the existence of the variables $x_1, \ldots, x_m$ making true the conjunction. In particular, by enumerating all refutations of the goal, one could obtain all values for the variables satisfying the conjunction—that is, the answer to the query

$$\{\langle x_1, \ldots, x_m\rangle \mid A_1 \wedge \cdots \wedge A_n\}.$$

This is not the case when one allows arbitrary definite clauses rather than datalog rules, as illustrated in the following example.

Example 12.4.9 Consider the program

$$\begin{aligned} S(x, z) &\leftarrow G(x, z) \\ S(x, z) &\leftarrow G(x, y), S(y, z) \\ S(x, x) &\leftarrow \end{aligned}$$

that computes in S the *reflexive* transitive closure of graph G. This is a set of definite clauses but not a datalog program because of the last rule. However, resolution can be extended to (and is indeed in general presented for) definite clauses. Observe, for instance, that the goal $\leftarrow S(w, w)$ is refuted with a substitution that does not bind variable w to a constant.

SLD resolution is a technique that provides proofs of facts. One must be sure that it produces only correct proofs (soundness) and that it is powerful enough to prove all

true facts (completeness). To conclude this section, we demonstrate the soundness and completeness of SLD resolution for datalog programs.

We use the following lemma:

LEMMA 12.4.10 Let $g \equiv \leftarrow A_1, \ldots, A_i, \ldots, A_n$ and $r \equiv B_1 \leftarrow B_2, \ldots, B_m$ be a goal and a rule with no variables in common, and let

$$g' \equiv \leftarrow A_1, \ldots, A_{i-1}, B_2, \ldots, B_m, A_{i+1}, \ldots, A_n.$$

If $\theta g'$ is a resolvent of g with r using θ, then the formula r implies:

$$\begin{aligned} r' &\equiv \neg\theta g' \rightarrow \neg\theta g \\ &= \theta(A_1 \wedge \cdots \wedge A_{i-1} \wedge B_2 \wedge \cdots \wedge B_m \wedge A_{i+1} \wedge \cdots \wedge A_n) \rightarrow \theta(A_1 \wedge \cdots \wedge A_n). \end{aligned}$$

Proof Let $\mathbf{J}$ be an instance over $sch(P)$ satisfying r and let valuation ν be such that

$$\mathbf{J} \models \nu[\theta(A_1) \wedge \cdots \wedge \theta(A_{i-1}) \wedge \theta(B_2) \wedge \cdots \wedge \theta(B_m) \wedge \theta(A_{i+1}) \wedge \cdots \wedge \theta(A_n)].$$

Because

$$\mathbf{J} \models \nu[\theta(B_2) \wedge \cdots \wedge \theta(B_m)]$$

and $\mathbf{J} \models B_1 \leftarrow B_2, \ldots, B_m$, $\mathbf{J} \models \nu[\theta(B_1)]$. That is, $\mathbf{J} \models \nu[\theta(A_i)]$. Thus

$$\mathbf{J} \models \nu[\theta(A_1) \wedge \cdots \wedge \theta(A_n)].$$

Hence for each ν, $\mathbf{J} \models \nu r'$. Therefore $\mathbf{J} \models r'$. Thus each instance over $sch(P)$ satisfying r also satisfies r', so r implies r'. ■

Using this lemma, we have the following:

THEOREM 12.4.11 (Soundness of SLD resolution) Let $P_{\mathbf{I}}$ be a program and $g \equiv \leftarrow A_1, \ldots, A_n$ a goal. If there exists an SLD-refutation of g with $P_{\mathbf{I}}$ and mgu's $\theta_1, \ldots, \theta_q$, then $P_{\mathbf{I}}$ implies

$$\theta_1 \circ \cdots \circ \theta_q(A_1 \wedge \cdots \wedge A_n).$$

Proof Let $\mathbf{J}$ be some instance over $sch(P)$ satisfying $P_{\mathbf{I}}$. Let $g_0 = g, \ldots, g_q = \square$ be an SLD refutation of g with $P_{\mathbf{I}}$ and for each j, let g_j be a resolvent of g_{j-1} with some rule in $P_{\mathbf{I}}$ using some mgu θ_j. Then for each j, the rule that is used implies $\neg g_j \rightarrow \theta_j(\neg g_{j-1})$ by Lemma 12.4.10. Because $\mathbf{J}$ satisfies $P_{\mathbf{I}}$, for each j,

$$\mathbf{J} \models \neg g_j \rightarrow \theta_j(\neg g_{j-1}).$$

Clearly, this implies that for each j,

$$\mathbf{J} \models \theta_{j+1} \circ \cdots \circ \theta_q(\neg g_j) \rightarrow \theta_j \circ \cdots \circ \theta_q(\neg g_{j-1}).$$

By transitivity, this shows that

$$\mathbf{J} \models \neg g_q \rightarrow \theta_1 \circ \cdots \circ \theta_q(\neg g_0),$$

and so

$$\mathbf{J} \models \mathit{true} \rightarrow \theta_1 \circ \cdots \circ \theta_q(\neg g).$$

Thus $\mathbf{J} \models \theta_1 \circ \cdots \circ \theta_q(A_1 \wedge \cdots \wedge A_n)$. ■

We next prove the converse of the previous result (namely, the completeness of SLD resolution).

THEOREM 12.4.12 (Completeness of SLD resolution) Let $P_{\mathbf{I}}$ be a program and $g \equiv \leftarrow A_1, \ldots, A_n$ a goal. If $P_{\mathbf{I}}$ implies $\neg g$, then there exists a refutation of g with $P_{\mathbf{I}}$.

Proof Suppose that $P_{\mathbf{I}}$ implies $\neg g$. Consider the set $\mathit{ground}(P_{\mathbf{I}})$ of instantiations of rules in $P_{\mathbf{I}}$ with constants in $\mathit{adom}(P, \mathbf{I})$. Clearly, $\mathit{ground}(P_{\mathbf{I}})(\emptyset)$ is a model of $P_{\mathbf{I}}$, so it satisfies $\neg g$. Thus there exists a valuation θ of the variables in g such that $\mathit{ground}(P_{\mathbf{I}})(\emptyset)$ satisfies $\neg\theta g$. By Theorem 12.4.5, there exists a refutation of θg using $\mathit{ground}(P_{\mathbf{I}})$.

Let $g_0 = \theta g, \ldots, g_p = \square$ be that refutation. We show by induction on k that for each k in $[0, p]$,

(†) there exists a derivation $g'_0 = g, \ldots, g'_k$ with $P_{\mathbf{I}}$ such that $g_k = \theta_k g'_k$ for some θ_k.

For suppose that (†) holds for each k. Then for $k = p$, there exists a derivation $g'_1 = g, \ldots, g'_p$ with $P_{\mathbf{I}}$ such that $\square = g_p = \theta_p g'_p$ for some θ_p, so $g'_p = \square$. Therefore there exists a refutation of g with $P_{\mathbf{I}}$.

The basis of the induction holds because $g_0 = \theta g = \theta g'_0$. Now suppose that (†) holds for some k. The next step of the refutation consists of selecting some atom B of g_k and applying a rule r in $\mathit{ground}(P_{\mathbf{I}})$. In g'_k select the atom B' with location in g' corresponding to the location of B in g_k. Note that $B = \theta_k B'$. In addition, we know that there is rule $r'' = B'' \leftarrow A''_1 \ldots A''_n$ in $P_{\mathbf{I}}$ that has r for instantiation via some substitution θ'' (such a pair B', r'' exists although it may not be unique). As usual, we can assume that the variables in g'_k are disjoint from those in r''. Let $\theta_k \oplus \theta''$ be the substitution defined by $\theta_k \oplus \theta''(x) = \theta_k(x)$ if x is a variable in g'_k, and $\theta_k \oplus \theta''(x) = \theta''(x)$ if x is a variable in r''. Clearly, $\theta_k \oplus \theta''(B') = \theta_k \oplus \theta''(B'') = B$ so, by Lemma 12.4.8 (i), B' and B'' have some mgu θ. Let g'_{k+1} be the resolvent of g'_k with r'', B' using mgu θ. By the definition of mgu, there exists a substitution θ_{k+1} such that $\theta_k \oplus \theta'' = \theta \circ \theta_{k+1}$. Clearly, $\theta_{k+1}(g'_{k+1}) = g_{k+1}$ and (†) holds for $k + 1$. By induction, (†) holds for each k. ■

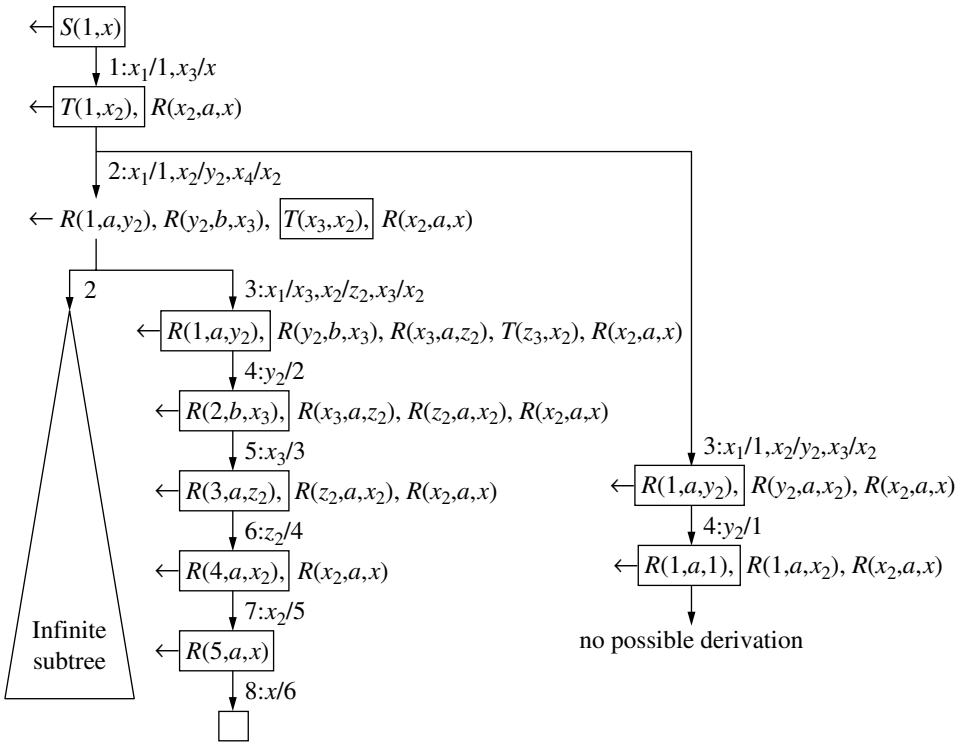


Figure 12.6: SLD tree

SLD Trees

We have shown that SLD resolution is sound and complete. Thus it provides an adequate top-down technique for obtaining the facts in the answer to a datalog program. To prove that a fact is in the answer, one must search for a refutation of the corresponding goal. Clearly, there are many refutations possible. There are two sources of nondeterminism in searching for a refutation: (1) the choice of the selected atom, and (2) the choice of the clause to unify with the atom. Now let us assume that we have fixed some golden rule, called a *selection rule*, for choosing which atom to select at each step in a refutation. A priori, such a rule may be very simple (e.g., as in Prolog, always take the leftmost atom) or in contrast very involved, taking into account the entire history of the refutation. Once an atom has been selected, we can systematically search for all possible unifying rules. Such a search can be represented in an *SLD tree*. For instance, consider the tree of Fig. 12.6 for the program in Example 12.4.2. The selected atoms are represented with boxes. Edges denote unifications used. Given $S(1, x)$, only one rule can be used. Given $T(1, x_2)$, two rules are applicable that account for the two descendants of vertex $T(1, x_2)$. The first number in edge labels denotes the rule that is used and the remaining part denotes the substitution. An SLD tree is a representation of all the derivations obtained with a fixed selection rule for atoms.

There are several important observations to be made about this particular SLD tree:

(i) It is successful because one branch yields □.

(ii) It has an infinite subtree that corresponds to an infinite sequence of applications of rule (2) of Example 12.4.2.

(iii) It has a blocking branch.

We can now explain (to a certain extent) the acronym SLD. SLD stands for selection rule-driven linear resolution for definite clauses. *Rule-driven* refers to the rule used for selecting the atom. An important fact is that the success or failure of an SLD tree does not depend on the rule for selecting atoms. This explains why the definition of an SLD tree does not specify the selection rule.

Datalog versus Logic Programming, Revisited

Having established the three semantics for datalog, we summarize briefly the main differences between datalog and the more general logic-programming (lp) framework.

Syntax: Datalog has only relation symbols, whereas lp uses also function symbols. Datalog requires variables in rule heads to appear in bodies; in particular, all unit clauses are ground.

Model-theoretic semantics: Due to the presence of function symbols in lp, models of lp programs may be infinite. Datalog programs always have finite models. Apart from this distinction, lp and datalog are identical with respect to model-theoretic semantics.

Fixpoint semantics: Again, the minimum fixpoint of the immediate consequence operator may be infinite in the lp case, whereas it is always finite for datalog. Thus the fixpoint approach does not necessarily provide a constructive semantics for lp.

Proof-theoretic semantics: The technique of SLD resolution is similar for datalog and lp, with the difference that the computation of mgu's becomes slightly more complicated with function symbols (see Exercise 12.20). For datalog, the significance of SLD resolution concerns primarily optimization methods inspired by resolution (such as "magic sets"; see Chapter 13). In lp, SLD resolution is more important. Due to the possibly infinite answers, the bottom-up approach of the fixpoint semantics may not be feasible. On the other hand, every fact in the answer has a finite proof by SLD resolution. Thus SLD resolution emerges as the practical alternative.

Expressive power: A classical result is that lp can express all recursively enumerable (r.e.) predicates. However, as will be discussed in Part E, the expressive power of datalog lies within PTIME. Why is there such a disparity? A fundamental reason is that function symbols are used in lp, and so an infinite domain of objects can be constructed from a finite set of symbols. Speaking technically, the result for lp states that if S is a (possibly infinite) r.e. predicate over terms constructed using a finite language, then there is an lp program that produces for some predicate symbol exactly the tuples in S. Speaking intuitively, this follows from the facts that viewed in a bottom-up sense, lp provides composition and looping, and terms of arbitrary length can be used as scratch paper

(e.g., to simulate a Turing tape). In contrast, the working space and output of range-restricted datalog programs are always contained within the active domain of the input and the program and thus are bounded in size.

Another distinction between lp and datalog in this context concerns the nature of expressive power results for datalog and for query languages in general. Specifically, a datalog program P is generally viewed as a mapping from instances of $edb(P)$ to instances of $idb(P)$. Thus expressive power of datalog is generally measured in comparison with mappings on families of database instances rather than in terms of expressing a single (possibly infinite) predicate.

12.5 Static Program Analysis

In this section, the static analysis of datalog programs is considered.[2] As with relational calculus, even simple static properties are undecidable for datalog programs. In particular, although tableau homomorphism allowed us to test the equivalence of conjunctive queries, equivalence of datalog programs is undecidable in general. This complicates a systematic search for alternative execution plans for datalog queries and yields severe limitations to query optimization. It also entails the undecidability of many other problems related to optimization, such as deciding when selection propagation (in the style of "pushing" selections in relational algebra) can be performed, or when parallel evaluation is possible.

We consider three fundamental static properties: satisfiability, containment, and a new one, boundedness. We exhibit a decision procedure for satisfiability. Recall that we showed in Chapter 5 that an analogous property is undecidable for CALC. The decidability of satisfiability for datalog may therefore be surprising. However, one must remember that, although datalog is more powerful than CALC in some respects (it has recursion), it is less powerful in others (there is no negation). It is the lack of negation that makes satisfiability decidable for datalog.

We prove the undecidability of containment and boundedness for datalog programs and consider variations or restrictions that are decidable.

Satisfiability

Let P be a datalog program. An intensional relation T is *satisfiable by* P if there exists an instance $\mathbf{I}$ over $edb(P)$ such that $P(\mathbf{I})(T)$ is nonempty. We give a simple proof of the decidability of satisfiability for datalog programs. We will soon see an alternative proof based on context-free languages.

We first consider constant-free programs. We then describe how to reduce the general case to the constant-free one.

To prove the result, we use an auxiliary result about instance homomorphisms that is of some interest in its own right. Note that any mapping θ from **dom** to **dom** can be extended to a homomorphism over the set of instances, which we also denote by θ.

---

[2] Recall that static program analysis consists of trying to detect statically (i.e., at compile time) properties of programs.

LEMMA 12.5.1 Let P be a constant-free datalog program, $\mathbf{I}, \mathbf{J}$ two instances over $sch(P)$, q a positive-existential query over $sch(P)$, and θ a mapping over **dom**. If $\theta(\mathbf{I}) \subseteq \mathbf{J}$, then (i) $\theta(q(\mathbf{I})) \subseteq q(\mathbf{J})$, and (ii) $\theta(P(\mathbf{I})) \subseteq P(\mathbf{J})$.

Proof For (i), observe that q is monotone and that $q \circ \theta \subseteq \theta \circ q$ (which is not necessary if q has constants). Because T_P can be viewed as a positive-existential query, a straightforward induction proves (ii). ■

This result does not hold for datalog programs with constants (see Exercise 12.21).

THEOREM 12.5.2 The satisfiability of an *idb* relation T by a constant-free datalog program P is decidable.

Proof Suppose that T is satisfiable by a constant-free datalog program P. We prove that $P(\mathbf{I}_a)(T)$ is nonempty for some particular instance $\mathbf{I}_a$. Let a be in **dom**. Let $\mathbf{I}_a$ be the instance over $edb(P)$ such that for each R in $edb(P)$, $\mathbf{I}_a(R)$ contains a single tuple with a in each entry. Because T is satisfiable by P, there exists $\mathbf{I}$ such that $P(\mathbf{I})(T) \neq \emptyset$. Consider the function θ that maps every constant in **dom** to a. Then $\theta(\mathbf{I}) \subseteq \mathbf{I}_a$. By the previous lemma, $\theta(P(\mathbf{I})) \subseteq P(\mathbf{I}_a)$. Therefore $P(\mathbf{I}_a)(T)$ is nonempty. Hence T is satisfiable by P iff $P(\mathbf{I}_a)(T) \neq \emptyset$. ■

Let us now consider the case of datalog programs *with* constants. Let P be a datalog program with constants. For example, suppose that b, c are the only two constants occurring in the program and that R is a binary relation occurring in P. We transform the problem into a problem without constants. Specifically, we replace R with nine new relations:

$$R_{\star\star},\ R_{b\star},\ R_{c\star},\ R_{\star b},\ R_{\star c},\ R_{bc},\ R_{cb},\ R_{bb},\ R_{cc}.$$

The first one is binary, the next four are unary, and the last four are 0-ary (i.e., are propositions). Intuitively, a fact $R(x, y)$ is represented by the fact $R_{\star\star}(x, y)$ if x, y are not in $\{b, c\}$; $R(b, x)$ with x not in $\{b, c\}$ is represented by $R_{b\star}(x)$, and similarly for $R_{c\star}$, $R_{\star b}$, $R_{\star c}$. The fact $R(b, c)$ is represented by proposition $R_{bc}()$, etc. Using this kind of transformation for each relation, one translates program P into a constant-free program P' such that T is satisfiable by P iff T_w is satisfiable by P' for some string w of $\star$ or constants occurring in P. (See Exercise 12.22a.)

Containment

Consider two datalog programs P, P' with the same extensional relations $edb(P)$ and a target relation T occurring in both programs. We say that P is *included* in P' *with respect to* T, denoted $P \subseteq_T P'$, if for each instance $\mathbf{I}$ over $edb(P)$, $P(\mathbf{I})(T) \subseteq P'(\mathbf{I})(T)$. The containment problem is undecidable. We prove this by reduction of the containment problem for context-free languages. The technique is interesting because it exhibits a correspondence between proof trees of certain datalog programs and derivation trees of context-free languages.

We first illustrate the correspondence in an example.

EXAMPLE 12.5.3 Consider the context-free grammar $G = (V, \Sigma, \Pi, S)$, where $V = \{S, T\}$, S is the start symbol, $\Sigma = \{a, b\}$, and the set Π of production rules is

$$S \to Ta$$
$$T \to abT \mid aa.$$

The corresponding datalog program P_G is the program of Example 12.4.1. A proof tree and its corresponding derivation tree are shown in Fig. 12.3.

We next formalize the correspondence between proof trees and derivation trees.
A context-free grammar is a ($\star$) grammar if the following hold:

(1) G is ϵ free (i.e., does not have any production of the form $X \to \epsilon$, where ϵ denotes the empty string) and

(2) the start symbol does not occur in any right-hand side of a production.

We use the following:

Fact It is undecidable, given ($\star$) grammars G_1, G_2, whether $L(G_1) \subseteq L(G_2)$.

For each ($\star$) grammar G, let P_G, the corresponding datalog program, be constructed (similar to Example 12.5.3) as follows: Let $G = (V, \Sigma, \Pi, S)$. We may assume without loss of generality that V is a set of relation names of arity 2 and Σ a set of elements from **dom**. Then $idb(P_G) = V$ and $edb(P_G) = \{R\}$, where R is a ternary relation. Let $x_1, x_2, \ldots$ be an infinite sequence of distinct variables. To each production in Π,

$$T \to C_1 \ldots C_n,$$

we associate a datalog rule

$$T(x_1, x_{n+1}) \leftarrow A_1, \ldots, A_n,$$

where for each i

- if C_i is a nonterminal T', then $A_i = T'(x_i, x_{i+1})$;
- if C_i is a terminal b, then $A_i = R(x_i, b, x_{i+1})$.

Note that, for any proof tree of a fact $S(a_1, a_n)$ using P_G, the sequence of its leaves is (in this order)

$$R(a_1, b_1, a_2), \ldots, R(a_{n-1}, b_{n-1}, a_n),$$

for some $a_2, \ldots, a_{n-1}$ and $b_1, \ldots, b_{n-1}$. The connection between derivation trees of G and proof trees of P_G is shown in the following.

Proposition 12.5.4 Let G be a ($\star$) grammar and P_G be the associated datalog program constructed as just shown. For each $a_1, \ldots, a_n, b_1, \ldots, b_{n-1}$, there is a proof tree of $S(a_1, a_n)$ from P_G with leaves $R(a_1, b_1, a_2), \ldots, R(a_{n-1}, b_{n-1}, a_n)$ (in this order) iff $b_1 \ldots b_{n-1}$ is in $L(G)$.

The proof of the proposition is left as Exercise 12.25. Now we can show the following:

Theorem 12.5.5 It is undecidable, given P, P' (with $edb(P) = edb(P')$) and T, whether $P \subseteq_T P'$.

Proof It suffices to show that

$$(\ddagger) \qquad \begin{array}{c} \text{for each pair } G_1, G_2 \text{ of } (\star) \text{ grammars,} \\ L(G_1) \subseteq L(G_2) \Leftrightarrow P_{G_1} \subseteq_S P_{G_2}. \end{array}$$

Suppose ($\ddagger$) holds and T containment is decidable. Then we obtain an algorithm to decide containment of ($\star$) grammars, which contradicts the aforementioned fact.

Let G_2, G_2 be two ($\star$) grammars. We show here that

$$L(G_1) \subseteq L(G_2) \Rightarrow P_{G_1} \subseteq_S P_{G_2}.$$

(The other direction is similar.) Suppose that $L(G_1) \subseteq L(G_2)$. Let $\mathbf{I}$ be over $edb(P_{G_1})$ and $S(a_1, a_n)$ be in $P_{G_1}(\mathbf{I})$. Then there exists a proof tree of $S(a_1, a_n)$ from P_{G_1} and $\mathbf{I}$, with leaves labeled by facts

$$R(a_1, b_1, a_2), \ldots, R(a_{n-1}, b_{n-1}, a_n),$$

in this order. By Proposition 12.5.4, $b_1 \ldots b_{n-1}$ is in $L(G_1)$. Because $L(G_1) \subseteq L(G_2)$, $b_1 \ldots b_{n-1}$ is in $L(G_2)$. By the proposition again, there is a proof tree of $S(a_1, a_n)$ from P_{G_2} with leaves $R(a_1, b_1, a_2), \ldots, R(a_{n-1}, b_{n-1}, a_n)$, all of which are facts in $\mathbf{I}$. Thus $S(a_1, a_n)$ is in $P_{G_2}(\mathbf{I})$, so $P_{G_1} \subseteq_S P_{G_2}$. ■

Note that the datalog programs used in the preceding construction are very particular: They are essentially chain programs. Intuitively, in a *chain program* the variables in a rule body form a chain. More precisely, rules in chain programs are of the form

$$A_0(x_0, x_n) \leftarrow A_1(x_0, x_1), A_2(x_1, x_2), \ldots, A_n(x_{n-1}, x_n).$$

The preceding proof can be tightened to show that containment is undecidable even for chain programs (see Exercise 12.26).

The connection with grammars can also be used to provide an alternate proof of the decidability of satisfiability; satisfiability can be reduced to the emptiness problem for context-free languages (see Exercise 12.22c).

Although containment is undecidable, there is a closely related, stronger property which is decidable—namely, *uniform containment*. For two programs P, P' over the same

set of intensional and extensional relations, we say that P is uniformly contained in P', denoted $P \subseteq P'$, iff for each $\mathbf{I}$ over $sch(P)$, $P(\mathbf{I}) \subseteq P'(\mathbf{I})$. Uniform containment is a sufficient condition for containment. Interestingly, one can decide uniform containment. The test for uniform containment uses dependencies studied in Part D and the fundamental *chase* technique (see Exercises 12.27 and 12.28).

Boundedness

A key problem for datalog programs (and recursive programs in general) is to estimate the depth of recursion of a given program. In particular, it is important to know whether for a given program the depth is bounded by a constant independent of the input. Besides being meaningful for optimization, this turns out to be an elegant mathematical problem that has received a lot of attention.

A datalog program P is *bounded* if there exists a constant d such that for each $\mathbf{I}$ over $edb(P)$, $stage(P, \mathbf{I}) \leq d$. Clearly, if a program is bounded it is essentially nonrecursive, although it may appear to be recursive syntactically. In some sense, it is falsely recursive.

EXAMPLE 12.5.6 Consider the following two-rule program:

$$Buys(x, y) \leftarrow Trendy(x), Buys(z, y) \qquad Buys(x, y) \leftarrow Likes(x, y)$$

This program is bounded because *Buys(z,y)* can be replaced in the body by *Likes(z,y)*, yielding an equivalent recursion-free program. On the other hand, the program

$$Buys(x, y) \leftarrow Knows(x, z), Buys(z, y) \qquad Buys(x, y) \leftarrow Likes(x, y)$$

is inherently recursive (i.e., is not equivalent to any recursion-free program).

It is important to distinguish truly recursive programs from falsely recursive (bounded) programs. Unfortunately, boundedness cannot be tested.

THEOREM 12.5.7 Boundedness is undecidable for datalog programs.

The proof is by reduction of the PCP (see Chapter 2). One can even show that boundedness remains undecidable under strong restrictions, such as that the programs that are considered (1) are constant-free, (2) contain a unique recursive rule, or (3) contain a unique intensional relation. Decidability results have been obtained for linear programs or chain-rule programs (see Exercise 12.31).

Bibliographic Notes

It is difficult to attribute datalog to particular researchers because it is a restriction or extension of many previously proposed languages; some of the early history is discussed in [MW88a]. The name *datalog* was coined (to our knowledge) by David Maier.

Many particular classes of datalog programs have been investigated. Examples are the class of *monadic* programs (all intensional relations have arity one), the class of *linear* programs (in the body of each rule of these programs, there can be at most one relation that is mutually recursive with the head relation; see Chapter 13), the class of *chain* programs [UG88, AP87a] (their syntax resembles that of context-free grammars), and the class of *single rule programs* or *sirups* [Kan88] (they consist of a single nontrivial rule and a trivial exit rule).

The fixpoint semantics that we considered in this chapter is due to [CH85]. However, it has been considered much earlier in the context of logic programming [vEK76, AvE82]. For logic programming, the existence of a least fixpoint is proved using [Tar55].

The study of stage functions $stage(d, H)$ is a major topic in [Mos74], where they are defined for finite structures (i.e., instances) as well as for infinite structures.

Resolution was originally proposed in the context of automatic theorem proving. Its foundations are due to Robinson [Rob65]. SLD resolution was developed in [vEK76]. These form the basis of logic programming introduced by Kowalski [Kow74] and [CKRP73] and led to the language Prolog. Nice presentations of the topic can be found in [Apt91, Llo87]. Standard SLD resolution is more general than that presented in this chapter because of the presence of function symbols. The development is similar except for the notion of unification, which is more involved. A survey of unification can be found in [Sie88, Kni89].

The programming language Prolog proposed by Colmerauer [CKRP73] is based on SLD resolution. It uses a particular strategy for searching for SLD refutations. Various ways to couple Prolog with a relational database system have been considered (see [CGT90]).

The undecidability of containment is studied in [CGKV88, Shm87]. The decidability of uniform containment is shown in [CK86, Sag88]. The decidability of containment for monadic programs is studied in [CGKV88]. The equivalence of recursive and nonrecursive datalog programs is shown to be decidable in [CV92]. The complexity of this problem is considered in [CV94].

Interestingly, bounded recursion is defined and used early in the context of universal relations [MUV84]. Example 12.5.6 is from [Nau86]. Undecidability results for boundedness of various datalog classes are shown in [GMSV87, GMSV93, Var88, Abi89]. Decidability results for particular subclasses are demonstrated in [Ioa85, Nau86, CGKV88, NS87, Var88].

Boundedness implies that the query expressed by the program is a positive existential query and therefore is expressible in CALC (over finite inputs). What about the converse? If infinite inputs are allowed, then (by a compactness argument) unboundedness implies nonexpressibility by CALC. But in the finite (database) case, compactness does not hold, and the question remained open for some time. Kolaitis observed that unboundedness does not imply nonexpressibility by CALC over finite structures for datalog with inequalities ($x \neq y$). (We did not consider comparators $\neq, <, \leq$, etc. in this chapter.) The question was settled by Ajtai and Gurevich [AG89], who showed by an elegant argument that no unbounded datalog program is expressible in CALC, even on finite structures.

Another decision problem for datalog concerns arises from the interaction of datalog with functional dependencies. In particular, it is undecidable, given a datalog program P,

set Σ of fd's on $edb(P)$, and set Γ of fd's on $idb(P)$ whether $P(\mathbf{I}) \models \Gamma$ whenever $\mathbf{I} \models \Sigma$ [AH88].

The expressive power of datalog has been investigated in [AC89, ACY91, CH85, Shm87, LM89, KV90c]. Clearly, datalog expresses only monotonic queries, commutes with homomorphisms of the database (if there are no constants in the program), and can be evaluated in polynomial time (see also Exercise 12.11). It is natural to wonder if datalog expresses *precisely* those queries. The answer is negative. Indeed, [ACY91] shows that the existence of a path whose length is a perfect square between two nodes is not expressible in datalog$^{\neq}$ (datalog augmented with inequalities $x \neq y$), and so not in datalog. This is a monotonic, polynomial-time query commuting with homomorphisms. The parallel complexity of datalog is surveyed in [Kan88].

The function symbols used in logic programming are interpreted over a Herbrand domain and are prohibited in datalog. However, it is interesting to incorporate arithmetic functions such as addition and multiplication into datalog. Such functions can also be viewed as infinite base relations. If these are present, it is possible that the bottom-up evaluation of a datalog program will not terminate. This issue was first studied in [RBS87], where *finiteness dependencies* were introduced. These dependencies can be used to describe how the finiteness of the range of a set of variables can imply the finiteness of the range of another variable. [For example, the relation $+(x, y, z)$ satisfies the finiteness dependencies $\{x, y\} \rightsquigarrow \{z\}$, $\{x, z\} \rightsquigarrow \{y\}$, and $\{y, z\} \rightsquigarrow \{x\}$.] Safety of datalog programs with infinite relations constrained by finiteness dependencies is undecidable [SV89]. Various syntactic conditions on datalog programs that ensure safety are developed in [RBS87, KRS88a, KRS88b, SV89]. Finiteness dependencies were used to develop a safety condition for the relational calculus with infinite base relations in [EHJ93]. Safety was also considered in the context of *data functions* (i.e., functions whose extent is predefined).

Exercises

Exercise 12.1 Refer to the Parisian **Metro** database. Give a datalog program that yields, for each pair of stations (a, b), the stations c such that c is reachable (1) from both a and b; and (2) from a or b.

Exercise 12.2 Consider a database consisting of the **Metro** and **Cinema** databases, plus a relation *Theater-Station* giving for each theater the closest metro station. Suppose that you live near the Odeon metro station. Write a program that answers the query "Near which metro station can I see a Bergman movie?" (Having spent many years in Los Angeles, you do not like walking, so your only option is to take the metro at Odeon and get off at the station closest to the theater.)

Exercise 12.3 (Same generation) Consider a binary relation *Child_of*, where the intended meaning of *Child_of*(a, b) is that a is the child of b. Write a datalog program computing the set of pairs (c, d), where c and d have a common ancestor and are of the same generation with respect to this ancestor.

Exercise 12.4 We are given two directed graphs G_{black} and G_{white} over the same set V of vertexes, represented as binary relations. Write a datalog program P that computes the set of pairs (a, b) of vertexes such that there exists a path from a to b where black and white edges alternate, starting with a white edge.

Exercise 12.5 Suppose we are given an undirected graph with colored vertexes represented by a binary relation *Color* giving the colors of vertexes and a binary relation *Edge* giving the connection between them. (Although *Edge* provides directed edges, we ignore the direction, so we treat the graph as undirected.) Say that a vertex is *good* if it is connected to a blue vertex (blue is a constant) or if it is connected to an excellent vertex. An *excellent* vertex is a vertex that is connected to an outstanding vertex and to a red vertex. An *outstanding* vertex is a vertex that is connected to a good vertex, an excellent one, and a yellow one. Write a datalog program that computes the excellent vertexes.

Exercise 12.6 Consider a directed graph G represented as a binary relation. Show a datalog program that computes a binary relation T containing the pairs (a, b) for which there is a path of odd length from a to b in G.

Exercise 12.7 Given a directed graph G represented as a binary relation, write a datalog program that computes the vertexes x such that (1) there exists a cycle of even length passing through x; (2) there is a cycle of odd length through x; (3) there are even- and odd-length cycles through x.

Exercise 12.8 Consider the following program P:

$$R(x, y) \leftarrow Q(y, x), S(x, y)$$
$$S(x, y) \leftarrow Q(x, y), T(x, z)$$
$$T(x, y) \leftarrow Q(x, z), S(z, y)$$

Let $\mathbf{I}$ be a relation over $edb(P)$. Describe the output of the program. Now suppose the first rule is replaced by $R(x, y) \leftarrow Q(y, x)$. Describe the output of the new program.

Exercise 12.9 Prove Lemma 12.3.1.

Exercise 12.10 Prove that datalog queries are monotone.

Exercise 12.11 Suppose P is some property of graphs definable by a datalog program. Show that P is preserved under extensions and homomorphisms. That is, if G is a graph satisfying P, then (1) every supergraph of G satisfies P and (2) if h is a graph homomorphism, then $h(G)$ satisfies P.

Exercise 12.12 Show that the following graph properties are not definable by datalog programs:

(i) The number of nodes is even.

(ii) There is a nontrivial cycle (a trivial cycle is an edge $\langle a, a \rangle$ for some vertex a).

(iii) There is a simple path of even length between two specified nodes.

Show that nontrivial cycles can be detected if inequalities of the form $x \neq y$ are allowed in rule bodies.

♠ **Exercise 12.13** [ACY91] Consider the query *perfect square* on graphs: Is there a path (not necessarily simple) between nodes a and b whose length is a perfect square?

(i) Prove that *perfect square* is preserved under extension and homomorphism.

(ii) Show that *perfect square* is not expressible in datalog.

Hint: For (ii), consider "words" consisting of simple paths from a to b, and prove a pumping lemma for words "accepted" by datalog programs.

Exercise 12.14 Present an algorithm that, given the set of proof trees of depth i with a program P and instance $\mathbf{I}$, constructs all proof trees of depth $i + 1$. Make sure that your algorithm terminates.

Exercise 12.15 Let P be a datalog program, $\mathbf{I}$ an instance of $edb(P)$, and R in $idb(P)$. Let u be a vector of distinct variables of the arity of R. Demonstrate that

$$P(\mathbf{I})(R) = \{\theta R(u) \mid \text{there is a refutation of} \leftarrow R(u) \text{ using } P_{\mathbf{I}} \text{ and substitutions } \theta_1, \ldots \theta_n \text{ such that } \theta = \theta_1 \circ \cdots \circ \theta_n\}.$$

Exercise 12.16 (Substitution lemma) Let $P_{\mathbf{I}}$ be a program, g a goal, and θ a substitution. Prove that if there exists an SLD refutation of θg with $P_{\mathbf{I}}$ and ν, there also exists an SLD refutation of g with $P_{\mathbf{I}}$ and $\theta \circ \nu$.

Exercise 12.17 Reprove Theorem 12.3.4 using Tarski's and Kleene's theorems stated in Remark 12.3.5.

Exercise 12.18 Prove the "if part" of Theorem 12.4.5.

Exercise 12.19 Prove Lemma 12.4.8.

★ **Exercise 12.20** (Unification with function symbols) In general logic programming, one can use function symbols in addition to relations. A *term* is then either a constant in **dom**, a variable in **var**, or an expression $f(t_1, \ldots, t_n)$, where f is an n-ary function symbol and each t_i is a term. For example, $f(g(x, 5), y, f(y, x, x))$ is a term. In this context, a substitution θ is a mapping from a subset of **var** into the set of terms. Given a substitution θ, it is extended in the natural manner to include all terms constructed over the domain of θ. Extend the definitions of unifier and mgu to terms and to atoms permitting terms. Give an algorithm to obtain the mgu of two atoms.

Exercise 12.21 Prove that Lemma 12.5.1 does not generalize to datalog programs with constants.

Exercise 12.22 This exercise develops three alternative proofs of the generalization of Theorem 12.5.2 to datalog programs with constants. Prove the generalization by

(a) using the technique outlined just after the statement of the theorem

(b) making a direct proof using as input an instance $\mathbf{I}_{C \cup \{a\}}$, where C is the set of all constants occurring in the program and a is new, and where each relation in $\mathbf{I}$ contains all tuples constructed using $C \cup \{a\}$

(c) reducing to the emptiness problem for context-free languages.

♠ **Exercise 12.23** (datalog$^{\neq}$) The language datalog$^{\neq}$ is obtained by extending datalog with a new predicate $\neq$ with the obvious meaning.

(a) Formally define the new language.

(b) Extend the least-fixpoint and minimal-model semantics to datalog$^{\neq}$.

★ (c) Show that satisfiability remains decidable for datalog$^{\neq}$ and that it can be tested in exponential time with respect to the size of the program.

★ **Exercise 12.24** Which of the properties in Exercise 12.12 are expressible in datalog$^{\neq}$?

Exercise 12.25 Prove Proposition 12.5.4.

Exercise 12.26 Prove that containment of chain datalog programs is undecidable. *Hint:* Modify the proof of Theorem 12.5.5 by using, for each $b \in \Sigma$, a relation R_b such that $R_b(x, y)$ iff $R(x, b, y)$.

Exercise 12.27 Prove that containment does not imply uniform containment by exhibiting two programs P, Q over the same *edb*'s and with S as common *idb* such that $P \subseteq_S Q$ but $P \not\subseteq Q$.

♠ **Exercise 12.28** (Uniform containment [CK86, Sag88]) Prove that uniform containment of two datalog programs is decidable.

Exercise 12.29 Prove that each nr-datalog program is bounded.

♠ **Exercise 12.30** [GMSV87, Var88] Prove Theorem 12.5.7. *Hint:* Reduce the halting problem of Turing machines on an empty tape to boundedness of datalog programs. More precisely, have the *edb* encode legal computations of a Turing machine on an empty tape, and have the program verify the correctness of the encoding. Then show that the program is unbounded iff there are unbounded computations of the machine on the empty tape.

Exercise 12.31 (Boundedness of chain programs) Prove decidability of boundedness for chain programs. *Hint:* Reduce testing for boundedness to testing for finiteness of a context-free language.

♠ **Exercise 12.32** This exercise demonstrates that datalog is likely to be stronger than positive first order extended by generalized transitive closure.

(a) [Coo74] Recall that a single rule program (sirup) is a datalog program with one nontrivial rule. Show that the sirup

$$R(x) \leftarrow R(y), R(z), S(x, y, z)$$

is complete in PTIME. (This has been called variously the graph accessibility problem and the blue-blooded water buffalo problem; a water buffalo is blue blooded only if both of its parents are.)

(b) [KP86] Show that the in some sense simpler sirup

$$R(x) \leftarrow R(y), R(z), T(y, x), T(x, z)$$

is complete in PTIME.

(c) [Imm87b] The *generalized transitive closure* operator is defined on relations with arity $2n$ so that $TC(R)$ is the output of the datalog program

$$\begin{aligned} ans(x_1, \ldots, x_{2n}) &\leftarrow R(x_1, \ldots, x_{2n}) \\ ans(x_1, \ldots, x_n, z_1, \ldots, z_n) &\leftarrow R(x_1, \ldots, x_n, y_1, \ldots, y_n), \\ &\qquad ans(y_1, \ldots, y_n, z_1, \ldots, z_n) \end{aligned}$$

Show that the positive first order extended with generalized transitive closure is in LOGSPACE.

13 Evaluation of Datalog

Alice: *I don't mean to sound naive, but isn't it awfully expensive to answer datalog queries?*

Riccardo: *Not if you use the right bag of tricks . . .*

Vittorio: *. . . and some magical wisdom.*

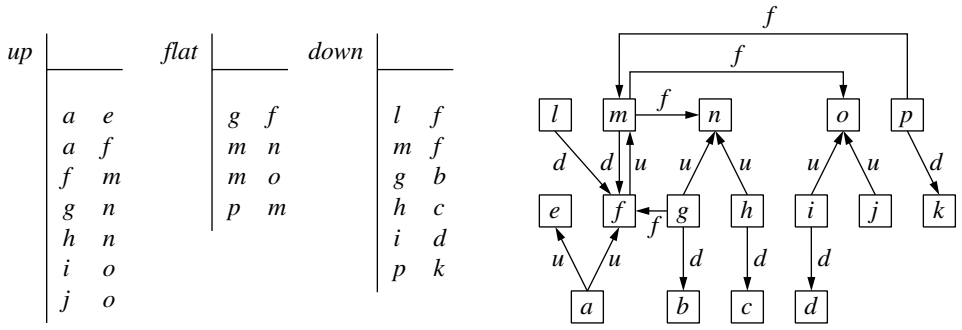
Sergio: *Well, there is no real need for magic. We will see that the evaluation is much easier if the algorithm knows where it is going and takes advantage of this knowledge.*

The introduction of datalog led to a flurry of research in optimization during the late 1980s and early 1990s. A variety of techniques emerged covering a range of different approaches. These techniques are usually separated into two classes depending on whether they focus on top-down or bottom-up evaluation. Another key dimension of the techniques concerns whether they are based on direct evaluation or propose some compilation of the query into a related query, which is subsequently evaluated using a direct technique.

This chapter provides a brief introduction to this broad family of heuristic techniques. A representative sampling of such techniques is presented. Some are centered around an approach known as "Query-Subquery"; these are top down and are based on direct evaluation. Others, centered around an approach called "magic set rewriting," are based on an initial preprocessing of the datalog program before using a fairly direct bottom-up evaluation strategy.

The advantage of top-down techniques is that selections that form part of the initial query can be propagated into the rules as they are expanded. There is no direct way to take advantage of this information in bottom-up evaluation, so it would seem that the bottom-up technique is at a disadvantage with respect to optimization. A rather elegant conclusion that has emerged from the research on datalog evaluation is that, surprisingly, there are bottom-up techniques that have essentially the same running time as top-down techniques. Exposition of this result is a main focus of this chapter.

Some of the evaluation techniques presented here are intricate, and our main emphasis is on conveying the essential ideas they use. The discussion is centered around the presentation of the techniques in connection with a concrete running example. In the cases of Query-Subquery and magic sets rewriting, we also informally describe how they can be applied in the general case. This is sufficient to give a precise understanding of the techniques without becoming overwhelmed by notation. Proofs of the correctness of these techniques are typically lengthy but straightforward and are left as exercises.



up	
a	*e*
a	*f*
f	*m*
g	*n*
h	*n*
i	*o*
j	*o*

flat	
g	*f*
m	*n*
m	*o*
p	*m*

down	
l	*f*
m	*f*
g	*b*
h	*c*
i	*d*
p	*k*

(a) The instance

(b) Represented as a graph

Figure 13.1: Instance $\mathbf{I}_0$ for RSG example

13.1 Seminaive Evaluation

The first stop on our tour of evaluation techniques is a strategy for improving the efficiency of the bottom-up technique described in Chapter 12. To illustrate this and the other techniques, we use as a running example the program "Reverse-Same-Generation" (RSG) given by

$$rsg(x, y) \leftarrow flat(x, y)$$
$$rsg(x, y) \leftarrow up(x, x1), rsg(y1, x1), down(y1, y)$$

and the sample instance $\mathbf{I}_0$ illustrated in Fig. 13.1. This is a fairly simple program, but it will allow us to present the main features of the various techniques presented throughout this chapter.

If the bottom-up algorithm of Chapter 12 is used to compute the value of *rsg* on input $\mathbf{I}_0$, the following values are obtained:

level 0: $\emptyset$
level 1: $\{\langle g, f\rangle, \langle m, n\rangle, \langle m, o\rangle, \langle p, m\rangle\}$
level 2: $\{\text{level 1}\} \cup \{\langle a, b\rangle, \langle h, f\rangle, \langle i, f\rangle, \langle j, f\rangle, \langle f, k\rangle\}$
level 3: $\{\text{level 2}\} \cup \{\langle a, c\rangle, \langle a, d\rangle\}$
level 4: $\{\text{level 3}\}$

at which point a fixpoint has been reached. It is clear that a considerable amount of redundant computation is done, because each layer recomputes all elements of the previous layer. This is a consequence of the monotonicity of the T_P operator for datalog programs P. This algorithm has been termed the *naive* algorithm for datalog evaluation. The central idea of the *seminaive* algorithm is to focus, to the extent possible, on the new facts generated at each level and thereby avoid recomputing the same facts.

Consider the facts inferred using the second rule of RSG in the consecutive stages of

the naive evaluation. At each stage, some new facts are inferred (until a fixpoint is reached). To infer a new fact at stage $i+1$, one must use at least one fact newly derived at stage i. This is the main idea of seminaive evaluation. It is captured by the following "version" of RSG, called RSG′:

$$\Delta^1_{rsg}(x, y) \leftarrow flat(x, y)$$
$$\Delta^{i+1}_{rsg}(x, y) \leftarrow up(x, x1), \Delta^i_{rsg}(y1, x1), down(y1, y)$$

where an instance of the second rule is included for each $i \geq 1$. Strictly speaking, this is not a datalog program because it has an infinite number of rules. On the other hand, it is not recursive.

Intuitively, Δ^i_{rsg} contains the facts in *rsg* newly inferred at the ith stage of the naive evaluation. To see this, we note a close relationship between the repeated applications of T_{RSG} and the values taken by the Δ^i_{rsg}. Let $\mathbf{I}$ be a fixed input instance. Then

- for $i \geq 0$, let $rsg^i = T^i_{RSG}(\mathbf{I})(rsg)$ (i.e., the value of *rsg* after i applications of T_{RSG} on $\mathbf{I}$); and
- for $i \geq 1$, let $\delta^i_{rsg} = RSG'(\mathbf{I})(\Delta^i_{rsg})$ (i.e., the value of Δ^i_{rsg} when $T_{RSG'}$ reaches a fixpoint on $\mathbf{I}$).

It is easily verified for each $i \geq 1$ that $T^{i-1}_{RSG'}(\mathbf{I})(\Delta^i_{rsg}) = \emptyset$ and $T^i_{RSG'}(\mathbf{I})(\Delta^i_{rsg}) = \delta^i_{rsg}$. Furthermore, for each $i \geq 0$ we have

$$rsg^{i+1} - rsg^i \subseteq \delta^{i+1}_{rsg} \subseteq rsg^{i+1}.$$

Therefore $\mathrm{RSG}(\mathbf{I})(rsg) = \cup_{1 \leq i}(\delta^i_{rsg})$. Furthermore, if j satisfies $\delta^j_{rsg} \subseteq \cup_{i<j}\delta^i_{rsg}$, then $\mathrm{RSG}(\mathbf{I})(rsg) = \cup_{i<j}\delta^i_{rsg}$, that is, only j levels of RSG′ need be computed to find $\mathrm{RSG}(\mathbf{I})(rsg)$. Importantly, bottom-up evaluation of RSG′ typically involves much less redundant computation than direct bottom-up evaluation of RSG.

Continuing with the informal development, we introduce now two refinements that further reduce the amount of redundant computation. The first is based on the observation that when executing RSG′, we do not always have $\delta^{i+1}_{rsg} = rsg^{i+1} - rsg^i$. Using $\mathbf{I}_0$, we have $\langle g, f \rangle \in \delta^2_{rsg}$ but not in $rsg^2 - rsg^1$. This suggests that the efficiency can be further improved by using $rsg^i - rsg^{i-1}$ in place of Δ^i_{rsg} in the body of the second "rule" of RSG′. Using a pidgin language that combines both datalog and imperative commands, the new version RSG″ is given by

$$\left\{ \begin{array}{lcl} \Delta^1_{rsg}(x, y) & \leftarrow & flat(x, y) \\ rsg^1 & := & \Delta^1_{rsg} \end{array} \right\}$$

$$\left\{ \begin{array}{lcl} temp^{i+1}_{rsg}(x, y) & \leftarrow & up(x, x1), \Delta^i_{rsg}(y1, x1), down(y1, y) \\ \Delta^{i+1}_{rsg} & := & temp^{i+1}_{rsg} - rsg^i \\ rsg^{i+1} & := & rsg^i \cup \Delta^{i+1}_{rsg} \end{array} \right\}$$

(where an instance of the second family of commands is included for each $i \geq 1$).

The second improvement to reduce redundant computation is useful when a given *idb* predicate occurs twice in the same rule. To illustrate, consider the nonlinear version of the ancestor program:

$$anc(x, y) \leftarrow par(x, y)$$
$$anc(x, y) \leftarrow anc(x, z), anc(z, y)$$

A seminaive "version" of this is

$$\left\{ \begin{array}{lcl} \Delta^1_{anc}(x, y) & \leftarrow & par(x, y) \\ anc^1 & := & \Delta^1_{anc} \end{array} \right\}$$

$$\left\{ \begin{array}{lcl} temp^{i+1}_{anc}(x, y) & \leftarrow & \Delta^i_{anc}(x, z), anc(z, y) \\ temp^{i+1}_{anc}(x, y) & \leftarrow & anc(x, z), \Delta^i_{anc}(z, y) \\ \Delta^{i+1}_{anc} & := & temp^{i+1}_{anc} - anc^i \\ anc^{i+1} & := & anc^i \cup \Delta^{i+1}_{anc} \end{array} \right\}$$

Note here that both Δ^i_{anc} and anc^i are needed to ensure that all new facts in the next level are obtained.

Consider now an input instance consisting of $par(1, 2), par(2, 3)$. Then we have

$$\Delta^1_{anc} = \{\langle 1, 2\rangle, \langle 2, 3\rangle\}$$
$$anc^1 = \{\langle 1, 2\rangle, \langle 2, 3\rangle\}$$
$$\Delta^2_{anc} = \{\langle 1, 3\rangle\}$$

Furthermore, both of the rules for $temp^2_{anc}$ will compute the join of tuples $\langle 1, 2\rangle$ and $\langle 2, 3\rangle$, and so we have a redundant computation of $\langle 1, 3\rangle$. Examples are easily constructed where this kind of redundancy occurs for at an arbitrary level $i > 0$ (see Exercise 13.2).

An approach for preventing this kind of redundancy is to replace the two rules for $temp^{i+1}$ by

$$temp^{i+1}(x, y) \leftarrow \Delta^i_{anc}(x, z), anc^{i-1}(z, y)$$
$$temp^{i+1}(x, y) \leftarrow anc^i(x, z), \Delta^i_{anc}(z, y)$$

This approach is adopted below.

We now present the seminaive algorithm for the general case. Let P be a datalog program over *edb* **R** and *idb* **T**. Consider a rule

$$S(u) \leftarrow R_1(v_1), \ldots, R_n(v_n), T_1(w_1), \ldots, T_m(w_m)$$

in P, where the R_k's are *edb* predicates and the T_j's are *idb* predicates. Construct for each $j \in [1, m]$ and $i \geq 1$ the rule

$$temp_S^{i+1}(u) \leftarrow R_1(v_1), \ldots, R_n(v_n),$$
$$T_1^i(w_1), \ldots, T_{j-1}^i(w_{j-1}), \Delta_{T_j}^i(w_j), T_{j+1}^{i-1}(w_{j+1}), \ldots, T_m^{i-1}(w_m).$$

Let P_S^i represent the set of all i-level rules of this form constructed for the *idb* predicate S (i.e., the rules for $temp_S^{i+1}$, j in $[1, m]$).

Suppose now that $T_1, \ldots, T_l$ is a listing of the *idb* predicates of P that occur in the body of a rule defining S. We write

$$P_S^i(\mathbf{I}, T_1^{i-1}, \ldots, T_l^{i-1}, T_1^i, \ldots, T_l^i, \Delta_{T_1}^i, \ldots, \Delta_{T_l}^i)$$

to denote the set of tuples that result from applying the rules in P_S^i to given values for input instance $\mathbf{I}$ and for the T_j^{i-1}, T_j^i, and $\Delta_{T_j}^i$.

We now have the following:

ALGORITHM 13.1.1 (Basic Seminaive Algorithm)

Input: Datalog program P and input instance $\mathbf{I}$

Output: $P(\mathbf{I})$

1. Set P' to be the rules in P with no *idb* predicate in the body;
2. $S^0 := \emptyset$, for each *idb* predicate S;
3. $\Delta_S^1 := P'(\mathbf{I})(S)$, for each *idb* predicate S;
4. $i := 1$;
5. do begin
   for each *idb* predicate S, where $T_1, \ldots, T_l$
   are the *idb* predicates involved in rules defining S,
   begin
   $S^i := S^{i-1} \cup \Delta_S^i$;
   $\Delta_S^{i+1} := P_S^i(\mathbf{I}, T_1^{i-1}, \ldots, T_l^{i-1}, T_1^i, \ldots, T_l^i, \Delta_{T_1}^i, \ldots, \Delta_{T_l}^i) - S^i$;
   end;
   $i := i + 1$
   end
   until $\Delta_S^i = \emptyset$ for each *idb* predicate S.
6. $s := s^i$, for each *idb* predicate S. ■

The correctness of this algorithm is demonstrated in Exercise 13.3. However, it is still doing a lot of unnecessary work on some programs. We now analyze the structure of datalog programs to develop an improved version of the seminaive algorithm. It turns out that this analysis, with simple control of the computation, allows us to know in advance which predicates are likely to grow at each iteration and which are not, either because they are already saturated or because they are not yet affected by the computation.

Let P be a datalog program. Form the *precedence graph* G_P for P as follows: Use the *idb* predicates in P as the nodes and include edge (R, R') if there is a rule with head predicate R' in which R occurs in the body. P is *recursive* if G_P has a directed cycle. Two predicates R and R' are *mutually recursive* if $R = R'$ or R and R' participate in the same

cycle of G_P. Mutual recursion is an equivalence relation on the *idb* predicates of P, where each equivalence class corresponds to a strongly connected component of G_P. A rule of P is *recursive* if the body involves a predicate that is mutually recursive with the head.

We now have the following:

ALGORITHM 13.1.2 (Improved Seminaive Algorithm)

Input: Datalog program P and edb instance $\mathbf{I}$

Output: $P(\mathbf{I})$

1. Determine the equivalence classes of *idb(P)* under mutual recursion.
2. Construct a listing $[R_1], \ldots, [R_n]$ of the equivalence classes, according to a topological sort of G_P (i.e., so that for each pair $i < j$ there is no path in G_P from R_j to R_i).
3. For $i = 1$ to n do
   Apply Basic Seminaive Algorithm to compute the values of predicates in $[R_i]$, treating all predicates in $[R_j]$, $j < i$, as *edb* predicates. ■

The correctness of this algorithm is left as Exercise 13.4.

Linear Datalog

We conclude this discussion of the seminaive approach by introducing a special class of programs.

Let P be a program. A rule in P with head relation R is *linear* if there is at most one atom in the body of the rule whose predicate is mutually recursive with R. P is *linear* if each rule in P is linear. We now show how the Improved Seminaive Algorithm can be simplified for such programs.

Suppose that P is a linear program, and

$$\rho : R(u) \leftarrow T_1(v_1), \ldots, T_n(v_n)$$

is a rule in P, where T_j is mutually recursive with R. Associate with this the "rule"

$$\Delta_R^{i+1}(u) \leftarrow T_1(v_1), \ldots, \Delta_{T_j}^i(v_j), \ldots, T_n(v_n).$$

Note that this is the only rule that will be associated by the Improved Seminaive Algorithm with ρ. Thus, given an equivalence class $[T_k]$ of mutually recursive predicates of P, the rules for predicates S in $[T_k]$ use only the Δ_S^i, but not the S^i. In contrast, as seen earlier, both the Δ_S^i and S^i must be used in nonlinear programs.

13.2 Top-Down Techniques

Consider the RSG program from the previous section, augmented with a selection-based query:

$$rsg(x, y) \leftarrow flat(x, y)$$
$$rsg(x, y) \leftarrow up(x, x1), rsg(y1, x1), down(y1, y)$$
$$query(y) \leftarrow rsg(a, y)$$

where a is a constant. This program will be called the *RSG query*. Suppose that seminaive evaluation is used. Then each pair of *rsg* will be produced, including those that are not used to derive any element of *query*. For example, using $\mathbf{I}_0$ of Fig. 13.1 as input, fact $rsg(f, k)$ will be produced but not used. A primary motivation for the top-down approaches to datalog query evaluation is to avoid, to the extent possible, the production of tuples that are not needed to derive any answer tuples.

For this discussion, we define a *datalog query* to be a pair (P, q), where P is a datalog program and q is a datalog rule using relations of P in its body and the new relation *query* in its head. We generally assume that there is only one rule defining the predicate *query*, and it has the form

$$query(u) \leftarrow R(v)$$

for some *idb* predicate R.

A fact is *relevant* to query (P, q) on input $\mathbf{I}$ if there is a proof tree for *query* in which the fact occurs. A straightforward criterion for improving the efficiency of any datalog evaluation scheme is to infer only relevant facts. The evaluation procedures developed in the remainder of this chapter attempt to satisfy this criterion; but, as will be seen, they do not do so perfectly.

The top-down approaches use natural heuristics to focus attention on relevant facts. In particular, they use the framework provided by SLD resolution. The starting point for these algorithms (namely, the query to be answered) often includes constants; these have the effect of restricting the search for derivation trees and thus the set of facts produced. In the context of databases without function symbols, the top-down datalog evaluation algorithms can generally be forced to terminate on all inputs, even when the corresponding SLD-resolution algorithm does not. In this section, we focus primarily on the query-subquery (QSQ) framework.

There are four basic elements of this framework:

1. Use the general framework of SLD resolution, but do it set-at-a-time. This permits the use of optimized versions of relational algebra operations.
2. Beginning with the constants in the original query, "push" constants from goals to subgoals, in a manner analogous to pushing selections into joins.
3. Use the technique of "sideways information passing" (see Chapter 6) to pass constant binding information from one atom to the next in subgoals.
4. Use an efficient global flow-of-control strategy.

Adornments and Subqueries

Recall the RSG query given earlier. Consider an SLD tree for it. The child of the root would be $rsg(a, y)$. Speaking intuitively, not all values for *rsg* are requested, but rather only those

with first coordinate a. More generally, we are interested in finding derivations for *rsg* where the first coordinate is *bound* and the second coordinate is *free*. This is denoted by the expression rsg^{bf}, where the superscript 'bf' is called an *adornment*.

The next layer of the SLD tree will have a node holding $flat(a, y)$ and a node holding $up(a, x1)$, $rsg(y1, x1)$, $down(y1, y)$. Answers generated for the first of these nodes are given by $\pi_2(\sigma_{1 = \text{'a'}}(flat))$. Answers for the other node can be generated by a left-to-right evaluation. First the set of possible values for $x1$ is $J = \pi_2(\sigma_{1 = \text{'a'}}(up))$. Next the possible values for $y1$ are given by $\{y1 \mid \langle y1, x1\rangle \in rsg \text{ and } \langle x1\rangle \in J\}$ (i.e., the first coordinate values of *rsg* stemming from second coordinate values in J). More generally, then, this calls for an evaluation of rsg^{fb}, where the second coordinate values are bound by J. Finally, given $y1$ values, these can be used with *down* to obtain y values (i.e., answers to the query).

As suggested by this discussion, a top-down evaluation of a query in which constants occur can be broken into a family of "subqueries" having the form (R^γ, J), where γ is an adornment for *idb* predicate R, and J is a set of tuples that give values for the columns bound by γ. Expressions of the form (R^γ, J) are called *subqueries*. If the RSG query were applied to the instance of Fig. 13.1, the first subquery generated would be $(rsg^{fb}, \{\langle e\rangle, \langle f\rangle\})$. As we shall see, the QSQ framework is based on a systematic evaluation of subqueries.

Let P be a datalog program and $\mathbf{I}$ an input instance. Suppose that R is an *idb* predicate and γ is an adornment for R (i.e., a string of b's and f's having length the arity of R). Then $bound(R, \gamma)$ denotes the coordinates of R bound in γ. Let t be a tuple over $bound(R, \gamma)$. Then a *completion* for t in R^γ is a tuple s such that $s[bound(R, \gamma)] = t$ and $s \in P(\mathbf{I})(R)$. The *answer* to a subquery (R^γ, J) over $\mathbf{I}$ is the set of all completions of all tuples in J.

The use of adornments within a rule body is a generalization of the technique of sideways information passing discussed in Chapter 6. Consider the rule

$$(*) \qquad R(x, y, z) \leftarrow R_1(x, u, v), R_2(u, w, w, z), R_3(v, w, y, a).$$

Suppose that a subquery involving R^{bfb} is invoked. Assuming a left-to-right evaluation, this will lead to subqueries involving R_1^{bff}, R_2^{bffb}, and R_3^{bbfb}. We sometimes rewrite the rule as

$$R^{bfb}(x, y, z) \leftarrow R_1^{bff}(x, u, v), R_2^{bffb}(u, w, w, z), R_3^{bbfb}(v, w, y, a)$$

to emphasize the adornments. This is an example of an *adorned rule*. As we shall see, the adornments of *idb* predicates in rule bodies shall be used to guide evaluations of queries and subqueries. It is common to omit the adornments of *edb* predicates.

The general algorithm for adorning a rule, given an adornment for the head and an ordering of the rule body, is as follows: (1) All occurrences of each bound variable in the rule head are bound, (2) all occurrences of constants are bound, and (3) if a variable x occurs in the rule body, then all occurrences of x in subsequent literals are bound. A different ordering of the rule body would yield different adornments. In general, we permit different orderings of rule bodies for different adornments of a given rule head. (A generalization of this technique is considered in Exercise 13.19.)

The definition of adorned rule also applies to situations in which there are repeated

variables or constants in the rule head (see Exercise 13.9). However, adornments do not capture all of the relevant information that can arise as the result of repeated variables or constants that occur in *idb* predicates in rule bodies. Mechanisms for doing this are discussed in Section 13.4.

Supplementary Relations and QSQ Templates

A key component of the QSQ framework is the use of *QSQ templates* which store appropriate information during intermediate stages of an evaluation. Consider again the preceding rule (*), and imagine attempting to evaluate the subquery (R^{bfb}, J). This will result in calls to the generalized queries $(R_1^{bff}, \pi_1(J))$, (R_2^{bffb}, K), and (R_3^{bbfb}, L) for some relations K and L that depend on the evaluation of the preceding queries. Importantly, note that relation K relies on values passed from both J and R_1, and L relies on values passed from R_1 and R_2. A QSQ template provides data structures that will remember all of the values needed during a left-to-right evaluation of a subquery.

To do this, QSQ templates rely on *supplementary relations*. A total of $n + 1$ supplementary relations are associated to a rule body with n atoms. For example, the supplementary relations $sup_0, \ldots, sup_3$ for the rule (*) with head adorned by R^{bfb} are

$$\begin{array}{l} R^{bfb}(x, y, z) \leftarrow R_1^{bff}(x, u, v), \;\; R_2^{bffb}(u, w, w, z), \;\; R_3^{bbfb}(v, w, y, a) \\ \qquad\qquad\quad \uparrow \qquad\qquad\quad \uparrow \qquad\qquad\quad \uparrow \qquad\qquad\quad \uparrow \\ \qquad\qquad sup_0[x, z] \quad sup_1[x, z, u, v] \quad sup_2[x, z, v, w] \quad sup_3[x, y, z] \end{array}$$

Note that variables serve as attribute names in the supplementary relations. Speaking intuitively, the body of a rule may be viewed as a process that takes as input tuples over the bound attributes of the head and produces as output tuples over the variables (bound and free) of the head. This determines the attributes of the first and last supplementary relations. In addition, a variable (i.e., an attribute name) is in some supplementary relation if it is has been bound by some previous literal and if it is needed in the future by some subsequent literal or in the result.

More formally, for a rule body with atoms $A_1, \ldots, A_n$, the set of variables used as attribute names for the i^{th} supplementary relation is determined as follows:

- For the 0^{th} (i.e., zeroth) supplementary relation, the attribute set is the set X_0 of bound variables of the rule head; and for the last supplementary relation, the attribute set is the set X_n of variables in the rule head.
- For $i \in [1, n-1]$, the attribute set of the i^{th} supplementary relation is the set X_i of variables that occur both "before" X_i (i.e., occur in $X_0, A_1, \ldots, A_i$) and "after" X_i (i.e., occur in $A_{i+1}, \ldots, A_n, X_n$).

The *QSQ template* for an adorned rule is the sequence $(sup_0, \ldots, sup_n)$ of relation schemas for the supplementary relations of the rule. During the process of QSQ query evaluation, relation instances are assigned to these schemas; typically these instances repeatedly acquire new tuples as the algorithm runs. Figure 13.2 shows the use of QSQ templates in connection with the RSG query.

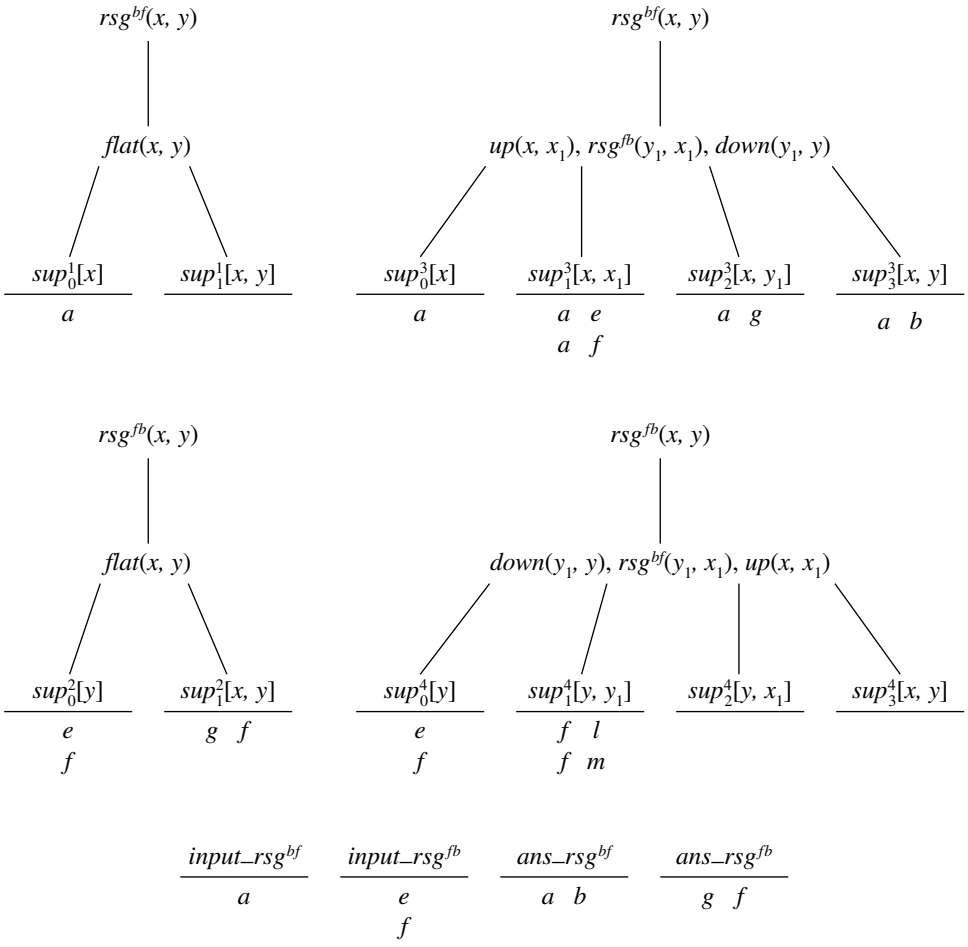


Figure 13.2: Illustration of QSQ framework

The Kernel of QSQ Evaluation

The key components of QSQ evaluation are as follows. Let (P, q) be a datalog query and let $\mathbf{I}$ be an *edb* instance. Speaking conceptually, QSQ evaluation begins by constructing an adorned rule for each adornment of each *idb* predicate in P and for the query q. In practice, the construction of these adorned rules can be lazy (i.e., they can be constructed only if needed during execution of the algorithm). Let (P^{ad}, q^{ad}) denote the result of this transformation.

The relevant adorned rules for the RSG query are as follows:

1. $rsg^{bf}(x, y) \leftarrow flat(x, y)$
2. $rsg^{fb}(x, y) \leftarrow flat(x, y)$

3. $rsg^{bf}(x, y) \leftarrow up(x, x1),\ rsg^{fb}(y1, x1),\ down(y1, y)$
4. $rsg^{fb}(x, y) \leftarrow down(y1, y),\ rsg^{bf}(y1, x1),\ up(x, x1).$

Note that in the fourth rule, the literals of the body are ordered so that the binding of y in *down* can be "passed" via $y1$ to *rsg* and via $x1$ to *up*.

A QSQ template is constructed for each relevant adorned rule. We denote the j^{th} (counting from 0) supplementary relation of the i^{th} adorned rule as sup_j^i. In addition, the following relations are needed and will serve as variables in the QSQ evaluation algorithm:

(a) for each *idb* predicate R and relevant adornment γ the variable ans_R^γ, with same arity as R;
(b) for each *idb* predicate R and relevant adornment γ, the variable $input\_R^\gamma$ with same arity as $bound(R, \gamma)$ (i.e., the number of b's occurring in γ); and
(c) for each supplementary relation sup_j^i, the variable sup_j^i.

Intuitively, $input\_R^\gamma$ will be used to form subqueries $(R^\gamma, input\_R^\gamma)$. The completion of tuples in $input\_R^\gamma$ will go to ans_R^γ. Thus ans_R^γ will hold tuples that are in $P(\mathbf{I})(R)$ and were generated from subqueries based on R^γ.

A QSQ algorithm begins with the empty set for each of the aforementioned relations. The query is then used to initialize the process. For example, the rule

$$query(y) \leftarrow rsg(a, y)$$

gives the initial value of $\{\langle a \rangle\}$ to $input\_rsg^{bf}$. In general, this gives rise to the subquery $(R^\gamma, \{t\})$, where t is constructed using the set of constants in the initial query.

There are essentially four kinds of steps in the execution. Different possible orderings for these steps will be considered. The first of these is used to initialize rules.

(A) Begin evaluation of a rule: This step can be taken whenever there is a rule with head predicate R^γ and there are "new" tuples in a variable $input\_R^\gamma$ that have not yet been processed for this rule. The step is to add the "new" tuples to the 0^{th} supplementary relation for this rule. However, only "new" tuples that unify with the head of the rule are added to the supplementary relation. A "new" tuple in $input\_R^\gamma$ might fail to unify with the head of a rule defining R if there are repeated variables or constants in the rule head (see Exercise 13.9).

New tuples are generated in supplementary relations sup_j^i in two ways: Either some new tuples have been obtained for sup_{j-1}^i (case B); or some new tuples have been obtained for the *idb* predicate occurring between sup_{j-1}^i and sup_j^i (case C).

(B) Pass new tuples from one supplementary relation to the next: This step can be taken whenever there is a set T of "new" tuples in a supplementary variable sup_{j-1}^i that have not yet been processed, and sup_{j-1}^i is not the last supplementary relation of the corresponding rule. Suppose that A_j is the atom in the rule immediately following sup_{j-1}^i.

Two cases arise:

(i) A_j is $R^\gamma(u)$ for some *edb* predicate R. Then a combination of joins and projections on R and T is used to determine the appropriate tuples to be added to sup_j^i.

(ii) A_j is $R^\gamma(u)$ for some *idb* predicate R. Note that each of the bound variables in γ occurs in sup_{j-1}^i. Two actions are now taken.
   (a) A combination of joins and projections on ans_R^γ (the current value for R) and T is used to determine the set T' of tuples to be added to sup_j^i.
   (b) The tuples in $T[bound(R, \gamma)] - input\_R^\gamma$ are added to $input\_R^\gamma$.

(C) Use new idb *tuples to generate new supplementary relation tuples:* This step is similar to the previous one but is applied when "new" tuples are added to one of the *idb* relation variables ans_R^γ. In particular, suppose that some atom A_j with predicate R^γ occurs in some rule, with surrounding supplementary variables sup_{j-1}^i and sup_j^i. In this case, use join and projection on all tuples in sup_{j-1}^i and the "new" tuples of ans_R^γ to create new tuples to be added to sup_j^i.

(D) Process tuples in the final supplementary relation of a rule: This step is used to generate tuples corresponding to the output of rules. It can be applied when there are "new" tuples in the final supplementary variable sup_n^i of a rule. Suppose that the rule predicate is R^γ. Add the new tuples in sup_n^i to ans_R^γ.

Example 13.2.1 Figure 13.2 illustrates the data structures and "scratch paper" relations used in the QSQ algorithm, in connection with the RSG query, as applied to the instance of Fig. 13.1. Recall the adorned version of the RSG query presented on page 321. The QSQ templates for these are shown in Fig. 13.2. Finally, the scratch paper relations for the *input*- and *ans*-variables are shown.

Figure 13.2 shows the contents of the relation variables after several steps of the QSQ approach have been applied. The procedure begins with the insertion of $\langle a \rangle$ into $input\_rsg^{bf}$; this corresponds to the rule

$$query(y) \leftarrow rsg(a, y)$$

Applications of step (A) place $\langle a \rangle$ into the supplementary variables sup_0^1 and sup_0^3. Step (B.i) then yields $\langle a, e \rangle$ and $\langle a, f \rangle$ in sup_1^3. Because ans_rsg^{fb} is empty at this point, step (B.ii.a) does not yield any tuples for sup_2^3. However, step (B.ii.b) is used to insert $\langle e \rangle$ and $\langle f \rangle$ into $input\_rsg^{fb}$. Application of steps (B) and (D) on the template of the second rule yield $\langle g, f \rangle$ in ans_rsg^{fb}. Application of steps (C), (B), and (D) on the template of the third rule now yield the first entry in ans_rsg^{bf}. The reader is invited to extend the evaluation to its conclusion (see Exercise 13.10). The answer is obtained by applying $\pi_2\sigma_{1 = \text{'a'}}$ to the final contents of ans_rsg^{bf}.

Global Control Strategies

We have now described all of the basic building blocks of the QSQ approach: the use of QSQ templates to perform information passing both into rules and sideways through rule bodies, and the three classes of relations used. A variety of global control strategies can be used for the QSQ approach. The most basic strategy is stated simply: Apply steps (A) through (D) until a fixpoint is reached. The following can be shown (see Exercise 13.12):

THEOREM 13.2.2 Let (P, q) be a datalog query. For each input $\mathbf{I}$, any evaluation of QSQ on (P^{ad}, q^{ad}) yields the answer of (P, q) on $\mathbf{I}$.

We now present a more specific algorithm based on the QSQ framework. This algorithm, called *QSQ Recursive* (QSQR) is based on a recursive strategy. To understand the central intuition behind QSQR, suppose that step (B) described earlier is to be performed, passing from supplementary relation sup^i_{j-1} across an *idb* predicate R^γ to supplementary relation sup^i_j. This may lead to the introduction of new tuples into sup^i_j by step (B.ii.a) and to the introduction of new tuples into $input\_R^\gamma$ by step (B.ii.b). The essence of QSQR is that it now performs a recursive call to determine the R^γ values corresponding to the new tuples added to $input\_R^\gamma$, before applying step (B) or (D) to the new tuples placed into sup^i_j.

We present QSQR in two steps: first a subroutine and then the recursive algorithm itself. During processing in QSQR, the global state includes values for ans_R^γ and $input\_R^\gamma$ for each *idb* predicate R and relevant adornment γ. However, the supplementary relations are not global—local copies of the supplementary relations are maintained by each call of the subroutine.

Subroutine Process subquery on one rule

Input: A rule for adorned predicate R^γ, input instance $\mathbf{I}$, a QSQR "state" (i.e., set of values for the *input*- and *ans*-variables), and a set $T \subseteq input\_R^\gamma$. (Intuitively, the tuples in T have not been considered with this rule yet).

Action:

1. Remove from T all tuples that do not unify with (the appropriate coordinates of) the head of the rule.
2. Set $sup_0 := T$. [This is step (A) for the tuples in T.]
3. Proceed sideways across the body $A_1, \ldots, A_n$ of the rule to the final supplementary relation sup_n as follows:
   For each atom A_j
   (a) If A_j has *edb* predicate R', then apply step (B.i) to populate sup_j.
   (b) If A_j has *idb* predicate R'^δ, then apply step (B.ii) as follows:
   (i) Set $S := sup_{j-1}[bound(R', \delta)] - input\_R'^\delta$.
   (ii) Set $input\_R'^\delta := input\_R'^\delta \cup S$. [This is step (B.ii.b).]
   (iii) (Recursively) call algorithm QSQR on the query (R'^δ, S).

[This has the effect of invoking step (A) and its consequences for the tuples in S.]

(iv) Use sup_{j-1} and the current value of global variable $ans\_R'^{\delta}$ to populate sup_j. [This includes steps (B.ii.a) and (C).]

4. Add the tuples produced for sup_n into the global variable ans_R^{γ}. [This is step (D).]

The main algorithm is given by the following:

ALGORITHM 13.2.3 (QSQR)

Input: A query of the form (R^{γ}, T), input instance $\mathbf{I}$, and a QSQR "state" (i.e., set of values for the *input*- and *ans*-variables).

Procedure:

1. Repeat until no new tuples are added to any global variable:
   Call the subroutine to process subquery (R^{γ}, T) on each rule defining R. ■

Suppose that we are given the query

$$query(u) \leftarrow R(v)$$

Let γ be the adornment of R corresponding to v, and let T be the singleton relation corresponding to the constants in v. To find the answer to the query, the QSQR algorithm is invoked with input (R^{γ}, T) and the global state where $input\_R^{\gamma} = T$ and all other *input*- and *ans*-variables are empty. For example, in the case of the *rsg* program, the algorithm is first called with argument $(rsg^{bf}, \{\langle a \rangle\})$, and in the global state $input\_rsg^{bf} = \{\langle a \rangle\}$. The answer to the query is obtained by performing a selection and projection on the final value of ans_R^{γ}.

It is straightforward to show that QSQR is correct (Exercise 13.12).

13.3 Magic

An exciting development in the field of datalog evaluation is the emergence of techniques for bottom-up evaluation whose performance rivals the efficiency of the top-down techniques. This family of techniques, which has come to be known as "magic set" techniques, simulates the pushing of selections that occurs in top-down approaches. There are close connections between the magic set techniques and the QSQ algorithm. The magic set technique presented in this section simulates the QSQ algorithm, using a datalog program that is evaluated bottom up. As we shall see, the magic sets are basically those sets of tuples stored in the relations $input\_R^{\gamma}$ and sup_j^i of the QSQ algorithm. Given a datalog query (P, q), the magic set approach transforms it into a new query (P^m, q^m) that has two important properties: (1) It computes the same answer as (P, q), and (2) when evaluated using a bottom-up technique, it produces only the set of facts produced by top-down approaches

(s1.1)	$rsg^{bf}(x, y)$	$\leftarrow input\_rsg^{bf}(x), flat(x, y)$
(s2.1)	$rsg^{fb}(x, y)$	$\leftarrow input\_rsg^{fb}(y), flat(x, y)$
(s3.1)	$sup_1^3(x, x1)$	$\leftarrow input\_rsg^{bf}(x), up(x, x1)$
(s3.2)	$sup_2^3(x, y1)$	$\leftarrow sup_1^3(x, x1), rsg^{fb}(y1, x1)$
(s3.3)	$rsg^{bf}(x, y)$	$\leftarrow sup_2^3(x, y1), down(y1, y)$
(s4.1)	$sup_1^4(y, y1)$	$\leftarrow input\_rsg^{fb}(y), down(y1, y)$
(s4.2)	$sup_2^4(y, x1)$	$\leftarrow sup_1^4(y, y1), rsg^{bf}(y1, x1)$
(s4.3)	$rsg^{fb}(x, y)$	$\leftarrow sup_2^4(y, x1), up(x, x1)$
(i3.2)	$input\_rsg^{bf}(x1)$	$\leftarrow sup_1^3(x, x1)$
(i4.2)	$input\_rsg^{fb}(y1)$	$\leftarrow sup_1^4(y, y1)$
(seed)	$input\_rsg^{bf}(a)$	$\leftarrow$
(query)	$query(y)$	$\leftarrow rsg^{bf}(a, y)$

Figure 13.3: Transformation of RSG query using magic sets

such as QSQ. In particular, then, (P^m, q^m) incorporates the effect of "pushing" selections from the query into bottom-up computations, as if by magic.

We focus on a technique originally called "generalized supplementary magic"; it is perhaps the most general magic set technique for datalog in the literature. (An earlier form of magic is considered in Exercise 13.18.) The discussion begins by explaining how the technique works in connection with the RSG query of the previous section and then presents the general algorithm.

As with QSQ, the starting point for magic set algorithms is an adorned datalog query (P^{ad}, q^{ad}). Four classes of rules are generated (see Fig. 13.3). The first consists of a family of rules for each rule of the adorned program P^{ad}. For example, recall rule (3) (see p. 321) of the adorned program for the RSG query presented in the previous section:

$$rsg^{bf}(x, y) \leftarrow up(x, x1), rsg^{fb}(y1, x1), down(y1, y).$$

We first present a primitive family of rules corrresponding to that rule, and then apply some optimizations.

(s3.0') $sup_0^3(x) \leftarrow input\_rsg^{bf}(x)$

(s3.1') $sup_1^3(x, x1) \leftarrow sup_0^3(x), up(x, x1)$

(s3.2) $sup_2^3(x, y1) \leftarrow sup_1^3(x, x1), rsg^{fb}(y1, x1)$

(s3.3') $sup_3^3(x, y) \leftarrow sup_2^3(x, y1), down(y1, y)$

(S3.4') $rsg^{bf}(x, y) \leftarrow sup_3^3(x, y)$

Rule (s3.0') corresponds to step (A) of the QSQ algorithm; rules (s3.1') and (s3.3') correspond to step (B.i); rule (s3.2) corresponds to steps (B.ii.a) and (C); and rule (s3.4') corresponds to step (D). In the literature, the predicate $input\_rsg^{fb}$ has usually been denoted as $magic\_rsg^{fb}$ and sup_j^i as $supmagic_j^i$. We use the current notation to stress the connection with the QSQ framework. Note that the predicate rsg^{bf} here plays the role of ans_rsg^{bf} there.

As can be seen by the preceding example, the predicates sup_0^3 and sup_3^3 are essentially redundant. In general, if the i^{th} rule defines R^γ, then the predicate sup_0^i is eliminated, with $input\_R^\gamma$ used in its place to eliminate rule (3.0') and to form

(s3.1) $sup_1^3(x, x1) \leftarrow input\_rsg^{bf}(x), up(x, x1).$

Similarly, the predicate of the last supplementary relation can be eliminated to delete rule (s3.4') and to form

(s3.3) $rsg^{bf}(x, y) \leftarrow sup_2^3(x, y1), down(y1, y).$

Therefore the set of rules (s3.0') through (s3.4') may be replaced by (s3.1), (s3.2), and (s3.3). Rules (s4.1), (s4.2), and (s4.3) of Fig. 13.3 are generated from rule (4) of the adorned program for the RSG query (see p. 321). (Recall how the order of the body literals in that rule are reversed to pass bounding information.) Finally, rules (s1.1) and (s2.1) stem from rules (1) and (2) of the adorned program.

The second class of rules is used to provide values for the *input* predicates [i.e., simulating step (B.ii.b) of the QSQ algorithm]. In the RSG query, one rule for each of $input\_rsg^{bf}$ and $input\_rsg^{fb}$ is needed:

(i3.2) $input\_rsg^{bf}(x1) \leftarrow sup_1^3(x, x1)$

(i4.2) $input\_rsg^{fb}(y1) \leftarrow sup_1^4(y, y1).$

Intuitively, the first rule comes from rule (s3.2). In other words, it follows from the second atom of the body of rule (3) of the original adorned program (see p. 321). In general, an adorned rule with k *idb* atoms in the body will generate k *input* rules of this form.

The third and fourth classes of rules include one rule each; these initialize and conclude the simulation of QSQ, respectively. The first of these acts as a "seed" and is derived from the initial query. In the running example, the seed is

$$input\_rsg^{bf}(a) \leftarrow .$$

The second constructs the answer to the query; in the example it is

$$query(y) \leftarrow rsg^{bf}(a, y).$$

From this example, it should be straightforward to specify the magic set rewriting of an adorned query (P^{ad}, q^{ad}) (see Exercise 13.16a).

The example showed how the "first" and "last" supplementary predicates sup_0^3 and sup_4^3 were redundant with $input\_rsg^{bf}$ and rsg^{bf}, respectively, and could be eliminated. Another improvement is to merge consecutive sequences of *edb* atoms in rule bodies as follows. For example, consider the rule

$$\text{(i)} \qquad R^{\gamma}(u) \leftarrow R_1^{\gamma_1}(u_1), \ldots, R_n^{\gamma_n}(u_n)$$

and suppose that predicate R_k is the last *idb* relation in the body. Then rules $(si.k), \ldots, (si.n)$ can be replaced with

$$(si.k'') \qquad R^{\gamma}(u) \leftarrow sup_{k-1}^{i}(v_{k-1}), R_k^{\gamma_k}(u_k), R_{k+1}^{\gamma_{k+1}}(u_{k+1}), \ldots, R_n^{\gamma_n}(u_n).$$

For example, rules (s3.2) and (s3.3) of Fig. 13.3 can be replaced by

$$(s3.2'') \qquad rsg^{bf}(x, y) \leftarrow sup_1^3(x, x1), rsg^{fb}(y1, x1), down(y1, y).$$

This simplification can also be used within rules. Suppose that R_k and R_l are *idb* relations with only *edb* relations occurring in between. Then rules $(i.k), \ldots, (i.l-1)$ can be replaced with

$$(si.k'') \qquad sup_{l-1}^{i}(v_{l-1}) \leftarrow sup_{k-1}^{i}(v_{k-1}), R_k^{\gamma_k}(u_k), R_{k+1}^{\gamma_{k+1}}(u_{k+1}), \ldots, R_{l-1}^{\gamma_{l-1}}(u_{l-1}).$$

An analogous simplification can be applied if there are multiple *edb* predicates at the beginning of the rule body.

To summarize the development, we state the following (see Exercise 13.16):

THEOREM 13.3.1 Let (P, q) be a query, and let (P^m, q^m) be the query resulting from the magic rewriting of (P, q). Then

(a) The answer computed by (P^m, q^m) on any input instance **I** is identical to the answer computed by (P, q) on **I**.

(b) The set of facts produced by the Improved Seminaive Algorithm of (P^m, q^m) on input **I** is identical to the set of facts produced by an evaluation of QSQ on **I**.

13.4 Two Improvements

This section briefly presents two improvements of the techniques discussed earlier. The first focuses on another kind of information passing resulting from repeated variables and constants occurring in *idb* predicates in rule bodies. The second, called counting, is applicable to sets of data and rules having certain acyclicity properties.

Repeated Variables and Constants in Rule Bodies (by Example)

Consider the program P_r:

(1) $T(x, y, z) \leftarrow R(x, y, z)$

(2) $T(x, y, z) \leftarrow S(x, y, w), T(w, z, z)$

$query(y, z) \leftarrow T(1, y, z)$

Consider as input the instance $\mathbf{I}_1$ shown in Fig. 13.4(a). The data structures for a QSQ evaluation of this program are shown in Fig. 13.4(b). (The annotations '$2 = $3', '$2 = $3 = 4', etc., will be explained later.)

A magic set rewriting of the program and query yields

$$
\begin{aligned}
T^{bff}(x, y, z) &\leftarrow input\_T^{bff}(x), R(x, y, z)\\
sup_1^2(x, y, w) &\leftarrow input\_T^{bff}(x), S(x, y, w)\\
T^{bff}(x, y, z) &\leftarrow sup_1^2(x, y, w), T^{bff}(w, z, z)\\
input\_T^{bff}(w) &\leftarrow sup_1^2(x, y, w)\\
input\_T^{bff}(1) &\leftarrow\\
query(y, z) &\leftarrow T^{bff}(1, y, z).
\end{aligned}
$$

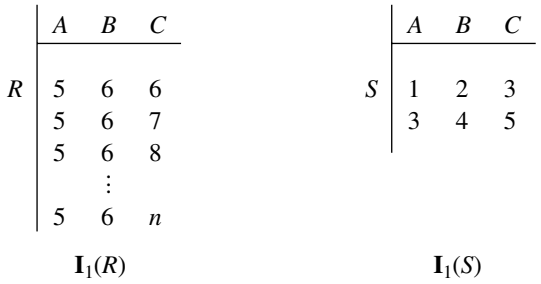
On input $\mathbf{I}_1$, the query returns the empty instance. Furthermore, the SLD tree for this query on $\mathbf{I}_1$ shown in Fig. 13.5, has only 9 goals and a total of 13 atoms, regardless of the value of n. However, both the QSQ and magic set approach generate a set of facts with size proportional to n (i.e., to the size of $\mathbf{I}_1$).

Why do both QSQ and magic sets perform so poorly on this program and query? The answer is that as presented, neither QSQ nor magic sets take advantage of restrictions on derivations resulting from the repeated z variable in the body of rule (2). Analogous examples can be developed for cases where constants appear in *idb* atoms in rule bodies.

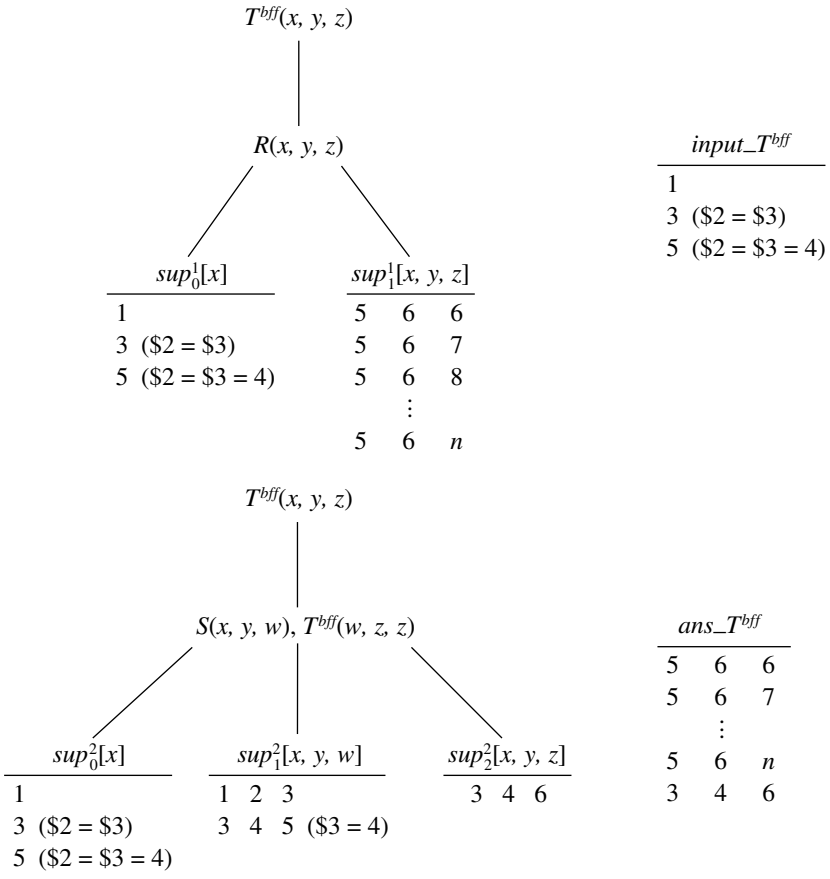
Both QSQ and magic sets can be enhanced to use such information. In the case of QSQ, the tuples added to supplementary relations can be annotated to carry information about restrictions imposed by the atom that "caused" the tuple to be placed into the leftmost supplementary relation. This is illustrated by the annotations in Fig. 13.4(b). First consider the annotation '$2 = $3' on the tuple $\langle 3\rangle$ in $input\_T^{bff}$. This tuple is included into $input\_T^{bff}$ because $\langle 1, 2, 3\rangle$ is in sup_1^2, and the next atom considered is $T^{bff}(w, z, z)$. In particular, then, any valid tuple (x, y, z) resulting from $\langle 3\rangle$ must have second and third coordinates equal. The annotation '$2 = $3' is passed with $\langle 3\rangle$ into sup_0^1 and sup_0^2.

Because variable y is bound to 4 in the tuple $\langle 3, 4, 5\rangle$ in sup_1^2, the annotation '$2 = $3' on $\langle 3\rangle$ in sup_0^2 "transforms" into '$3 = 4' on this new tuple. This, in turn, implies the annotation '$2 = $3 = 4' when $\langle 5\rangle$ is added to $input\_T^{bff}$ and to both sup_0^1 and sup_0^2.

Now consider the tuple $\langle 5\rangle$ in sup_0^1, with annotation ($2 = $3 = 4). This can generate a tuple in sup_1^1 only if $\langle 5, 4, 4\rangle$ is in R. For input $\mathbf{I}_1$ this tuple is not in R, and so the annotated



(a) Sample input instance $\mathbf{I}_1$



(b) QSQ evaluation

Figure 13.4: Behavior of QSQ on program with repeated variables

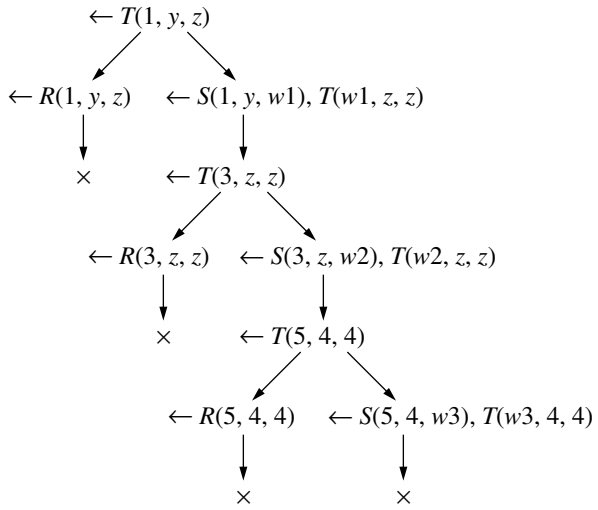


Figure 13.5: Behavior of SLD on program with repeated variables

tuple $\langle 5 \rangle$ in sup_0^1 generates nothing (even though in the original QSQ framework many tuples are generated). Analogously, because there is no tuple $\langle 5, 4, w \rangle$ in S, the annotated tuple $\langle 5 \rangle$ of sup_0^2 does not generate anything in sup_1^2. This illustrates how annotations can be used to restrict the facts generated during execution of QSQ.

More generally, annotations on tuples are conjunctions of equality terms of the form '$\$i = \$j$' and '$\$i = a$' (where a is a constant). During step (B.ii.b) of QSQ, annotations are associated with new tuples placed into relations $input\_R^\gamma$. We permit the same tuple to occur in $input\_R^\gamma$ with different annotations. This enhanced version of QSQ is called *annotated QSQ*. The enhancement correctly produces all answers to the initial query, and the set of facts generated now closely parallels the set of facts and assignments generated by the SLD tree corresponding to the QSQ templates used.

The magic set technique can also be enhanced to incorporate the information captured by the annotations just described. This is accomplished by an initial preprocessing of the program (and query) called "subgoal rectification." Speaking loosely, a subgoal corresponding to an *idb* predicate is *rectified* if it has no constants and no repeated variables. Rectified subgoals may be formed from nonrectified ones by creating new *idb* predicates that correspond to versions of *idb* predicates with repeated variables and constants. For example, the following is the result of rectifying the subgoals of the program P_r:

$$
\begin{aligned}
T(x, y, z) &\leftarrow R(x, y, z) \\
T(x, y, z) &\leftarrow S(x, y, w), T_{\$2=\$3}(w, z)
\end{aligned}
$$

$$
\begin{aligned}
T_{\$2=\$3}(x, z) &\leftarrow R(x, z, z) \\
T_{\$2=\$3}(x, z) &\leftarrow S(x, z, w), T_{\$2=\$3}(w, z)
\end{aligned}
$$

$$query(y, z) \leftarrow T(1, y, z)$$
$$query(z, z) \leftarrow T_{\$2=\$3}(1, z).$$

It is straightforward to develop an iterative algorithm that replaces an arbitrary datalog program and query with an equivalent one, all of whose *idb* subgoals are rectified (see Exercise 13.20). Note that there may be more than one rule defining the query after rectification.

The magic set transformation is applied to the rectified program to obtain the final result. In the preceding example, there are two relevant adornments for the predicate $T_{\$2=\$3}$ (namely, *bf* and *bb*).

The following can be verified (see Exercise 13.21):

THEOREM 13.4.1 (Informal) The framework of annotated QSQ and the magic set transformation augmented with subgoal rectification are both correct. Furthermore, the set of *idb* predicate facts generated by evaluating a datalog query with either of these techniques is identical to the set of facts occurring in the corresponding SLD tree.

A tight correspondence between the assignments in SLD derivation trees and the supplementary relations generated both by annotated QSQ and rectified magic sets can be shown. The intuitive conclusion drawn from this development is that top-down and bottom-up techniques for datalog evaluation have essentially the same efficiency.

Counting (by Example)

We now present a brief sketch of another improvement of the magic set technique. It is different from the previous one in that it works only when the underlying data set is known to have certain acyclicity properties.

Consider evaluating the following SG query based on the Same-Generation program:

(1) $sg(x, y) \leftarrow flat(x, y)$

(2) $sg(x, y) \leftarrow up(x, x1), sg(x1, y1), down(y1, y)$

$query(y) \leftarrow sg(a, y)$

on the input $\mathbf{J}_n$ given by

$$\mathbf{J}_n(up) = \{\langle a, b_i\rangle \mid i \in [1, n]\} \cup \{\langle b_i, c_j\rangle \mid i, j \in [1, n]\}$$
$$\mathbf{J}_n(flat) = \{\langle c_i, d_j\rangle \mid i, j \in [1, n]\}$$
$$\mathbf{J}_n(down) = \{\langle d_i, e_j\rangle \mid i, j \in [1, n]\} \cup \{\langle e_i, f\rangle \mid i \in [1, n]\}.$$

Instance $\mathbf{J}_2$ is shown in Fig. 13.6.

The completed QSQ template on input $\mathbf{J}_2$ for the second rule of the SG query is shown in Fig. 13.7(a). (The tuples are listed in the order in which QSQR would discover them.) Note that on input $\mathbf{J}_n$ both sup_1^2 and sup_2^2 would contain $n(n + 1)$ tuples.

Consider now the proof tree of SG having root $sg(a, f)$ shown in Fig. 13.8 (see Chapter 12). There is a natural correspondence of the children at depth 1 in this tree with the supplementary relation atoms $sup_0^2(a)$, $sup_1^2(a, b_1)$, $sup_2^2(a, e_1)$, and $sup_3^2(a, f)$ generated

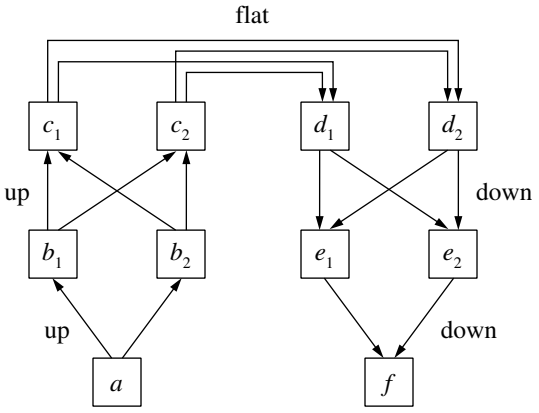


Figure 13.6: Instance $\mathbf{J}_2$ for counting

by QSQ; and between the children at depth 2 with $sup_0^2(b_1)$, $sup_1^2(b_1, c_1)$, $sup_2^2(b_1, d_1)$, and $sup_3^2(b_1, e_1)$.

A key idea in the counting technique is to record information about the depths at which supplementary relation atoms occur. In some cases, this permits us to ignore some of the specific constants present in the supplementary atoms. You will find that this is illustrated in Fig. 13.7(b). For example, we show atoms $count\_sup_0^2(1, a)$, $count\_sup_1^2(1, b_1)$, $count\_sup_2^2(1, e_1)$, and $count\_sup_3^2(1, f)$ that correspond to the supplementary atoms $sup_0^2(a)$, $sup_1^2(a, b_1)$, $sup_2^2(a, e_1)$, and $sup_3^2(a, f)$. Note that, for example, $count\_sup_1^2(2, c_1)$ corresponds to both $sup_1^2(b_1, c_1)$ and $sup_1^2(b_2, c_1)$.

More generally, the modified supplementary relation atoms hold an "index" that indicates a level in a proof tree corresponding to how the atom came to be created. Because of the structure of SG, and assuming that the *up* relation is acyclic, these modified supplementary relations can be used to find query answers. Note that on input $\mathbf{J}_n$, the relations $countsup_1^{2'}$ and $count\_sup_2^{2'}$ hold $2n$ tuples each rather than $n(n+1)$, as in the original QSQ approach.

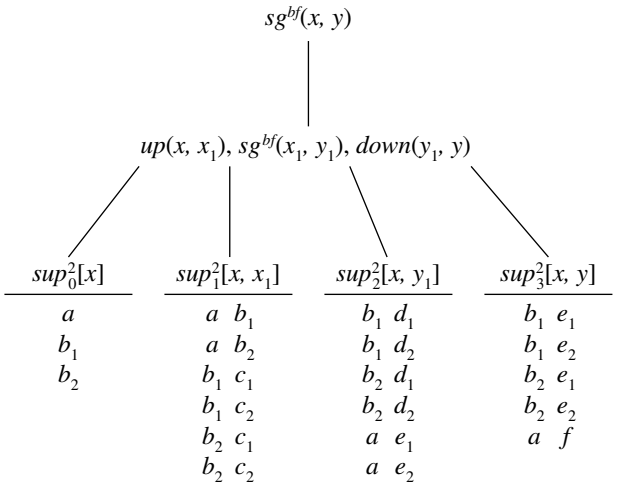
We now describe how the magic set program associated with the SG query can be transformed into an equivalent program (on acyclic input) that uses the indexes suggested by Fig. 13.7(b). The magic set rewriting of the SG query is given by

(s1.1) $sg^{bf}(x, y) \leftarrow input\_sg^{bf}(x), flat(x, y)$

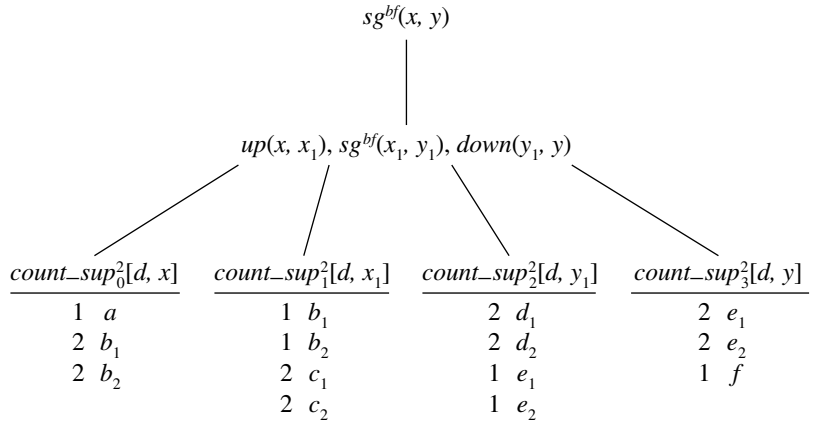
(s2.1) $sup_1^2(x, x1) \leftarrow input\_sg^{bf}(x), up(x, x1)$

(s2.2) $sup_2^2(x, y1) \leftarrow sup_1^2(x, x1), sg^{bf}(x1, y1)$

(s2.3) $sg^{bf}(x, y) \leftarrow sup_2^2(x, y1), down(y1, y)$



(a) Completed QSQ template for sg^{bf} on input $\mathbf{J}_2$



(b) Alternative QSQ "template," using indices

Figure 13.7: Illustration of intuition behind counting

(i2.2) $\quad input\_sg^{bf}(x1) \leftarrow sup_1^2(x, x1)$

(seed) $\quad input\_sg^{bf}(a) \leftarrow$

(query) $\quad query(y) \leftarrow sg^{bf}(a, y).$

The counting version of this is now given. (In other literature on counting, the seed is initialized with 0 rather than 1.)

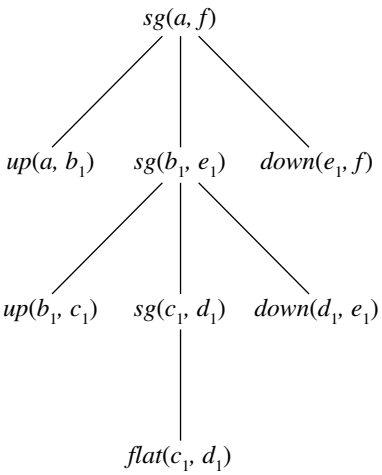


Figure 13.8: A proof tree for $sg(a, f)$

(c-s1.1) $count\_sg^{bf}(I, y) \leftarrow count\_input\_sg^{bf}(I, x), flat(x, y)$

(c-s2.1) $count\_sup_1^2(I, x1) \leftarrow count\_input\_sg^{bf}(I, x), up(x, x1)$

(c-s2.2) $count\_sup_2^2(I, y1) \leftarrow count\_sup_1^2(I, x1), count\_sg^{bf}(I + 1, y1)$

(c-s2.3) $count\_sg^{bf}(I, y) \leftarrow count\_sup_2^2(I, y1), down(y1, y)$

(c-i2.2) $count\_input\_sg^{bf}(I + 1, x1) \leftarrow count\_sup_1^2(I, x1)$

(c-seed) $count\_input\_sg^{bf}(1, a) \leftarrow$

(c-query) $query(y) \leftarrow count\_sg^{bf}(1, y)$

In the preceding, expressions such as $I + 1$ are viewed as a short hand for using a variable J in place of $I + 1$ and including $J = I + 1$ in the rule body.

In the counting version, the first coordinate of each supplementary relation keeps track of a level in a proof tree rather than a specific value. Intuitively, when "constructing" a sequence of supplementary atoms corresponding to a given level of a proof tree, each *idb* atom used must have been generated from the next deeper level. This is why $count\_sg^{bf}(I + 1, y1)$ is used in rule (c-s2.2). Furthermore, rule (c-i2.2) initiates the "construction" corresponding to a new layer of the proof tree.

The counting program of the preceding example is not safe, in the sense that on some inputs the program may produce an infinite set of tuples in some predicates (e.g., $count\_sup_1^2$). For example, this will happen if there is a cycle in the *up* relation reachable from a. Analogous situations occur with most applications of counting. As a result, the counting technique can only be used where the underlying data set is known to satisfy certain restrictions.

This preceding example is a simple application of the general technique of counting. A more general version of counting uses three kinds of indexes. The first, illustrated in the example, records information about levels of proof trees. The second is used to record information about what rule is being expanded, and the third is used to record which atom of the rule body is being considered (see Exercise 13.23). A description of the kinds of programs for which the counting technique can be used is beyond the scope of this book. Although limited in applicability, the counting technique has been shown to yield significant savings in some contexts.

Bibliographic Notes

This chapter has presented a brief introduction to the research on heuristics for datalog evaluation. An excellent survey of this work is [BR88a], which presents a taxonomy of different techniques and surveys a broad number of them. Several books provide substantial coverage of this area, including [Bid91a, CGT90, Ull89b]. Experimental results comparing several of the techniques in the context of datalog are described in [BR88b]. An excellent survey on deductive database systems, which includes an overview of several prototype systems that support datalog, is presented in [RU94].

The naive and seminaive strategies for datalog evaluation underlie several early investigations and implementations [Cha81b, MS81]; the seminaive strategy for evaluation is described in [Ban85, Ban86], which also propose various refinements. The use of T^{i-1} and T^i in Algorithm 13.1.1 is from [BR87b]. Reference [CGT90] highlights the close relationship of these approaches to the classical Jacobi and Gauss-Seidel algorithms of numerical analysis.

An essential ingredient of the top-down approaches to datalog evaluation is that of "pushing" selections into recursions. An early form of this was developed in [AU79], where selections and projections are pushed into restricted forms of fixpoint queries (see Chapter 14 for the definition of fixpoint queries).

The Query-Subquery (QSQ) approach was initially presented in [Vie86]; the independently developed method of "extension tables" [DW87] is essentially equivalent to this. The QSQ approach is extended in [Vie88, Vie89] to incorporate certain global optimizations. An extension of the technique to general logic programming, called SLD-AL, is developed in [Vie87a, Vie89]. Related approaches include APEX [Loz85], Earley Deduction [PW80, Por86], and those of [Nej87, Roe87]. The connection between context-free parsing and datalog evaluation is highlighted in [Lan88].

The algorithms of the QSQ family are sometimes called "memo-ing" approaches, because they use various data structures to remember salient inferred facts to filter the work of traditional SLD resolution.

Perhaps the most general of the top-down approaches uses "rule/goal" graphs [Ull85]; these potentially infinite trees intuitively correspond to a breadth-first, set-at-a-time execution of SLD resolution. Rule/goal graphs are applied in [Van86] to evaluate datalog queries in distributed systems. Similar graph structures have also been used in connection with general logic programs (e.g., [Kow75, Sic76]). A survey of several graph-based approaches is [DW85].

Turning to bottom-up approaches, the essentially equivalent approaches of [HN84] and

[GdM86] develop iterative algebraic programs for linear datalog programs. [GS87] extends these. A more general approach based on rewriting iterative algebra programs is presented in [CT87, Tan88].

The magic set and counting techniques originally appeared for linear datalog in [BMSU86]. Our presentation of magic sets is based on an extended version called "generalized supplementary magic sets" [BR87a, BR91]. That work develops a general notion of sideways information passing based on graphs (see Exercise 13.19), and develops both magic sets and counting in connection with general logic programming. The Alexander method [RLK86, Ker88], developed independently, is essentially the same as generalized supplementary magic sets for datalog. This was generalized to logic programming in [Sek89]. Magic set rewriting has also been applied to optimize SQL queries [MFPR90].

The counting method is generalized and combined with magic sets in [SZ86, SZ88]. Supplementary magic is incorporated in [BR91]. Analytic comparisons of magic and counting for selected programs are presented in [MSPS87].

Another bottom-up technique is Static Filtering [KL86a, KL86b]. This technique forms a graph corresponding to the flow of tuples through a bottom-up evaluation and then modifies the graph in a manner that captures information passing resulting from constants in the initial query.

Several of the investigations just mentioned, including [BR87a, KL86a, KL86b, Ull85, Vie86], emphasize the idea that sideways information passing and control are largely independent. Both [SZ88] and [BR91] describe fairly general mechanisms for specifying and using alternative sideways information passing and related message passing. A more general form of sideways information passing, which passes bounding inequalities between subgoals, is studied in [APP+86]. A formal framework for studying the success of pushing selections into datalog programs is developed in [BKBR87].

Several papers have studied the connection between top-down and bottom-up evaluation techniques. One body of the research in this direction focuses on the sets of facts generated by the top-down and bottom-up techniques. One of the first results relating top-down and bottom-up is from [BR87a, BR91], where it is shown that if a top-down technique and the generalized supplementary magic set technique use a given family of sideways information passing techniques, then the sets of intermediate facts produced by both techniques correspond. That research is conducted in the context of general logic programs that are range restricted. These results are generalized to possibly non-range-restricted logic programs in the independent research [Ram91] and [Sek89]. In that research, bottom-up evaluations may use terms and tuples that include variables, and bottom-up evaluation of rewritten programs uses unification rather than simple relational join. A close correspondence between top-down and bottom-up evaluation for datalog was established in [Ull89a], where subgoal rectification is used. The treatment of Program P_r and Theorem 13.4.1 are inspired by that development. This close correspondence is extended to arbitrary logic programs in [Ull89b]. Using a more detailed cost model, [SR93] shows that bottom-up evaluation asymptotically dominates top-down evaluation for logic programs, even if they produce nonground terms in their output.

A second direction of research on the connection between top-down and bottom-up approaches provides an elegant unifying framework [Bry89]. Recall in the discussion of Theorem 13.2.2 that the answer to a query can be obtained by performing the steps of

the QSQ until a fixpoint is reached. Note that the fixpoint operator used in this chapter is different from the conventional bottom-up application of T_P used by the naive algorithm for datalog evaluation. The framework presented in [Bry89] is based on meta-interpreters (i.e., interpreters that operate on datalog rules in addition to data); these can be used to specify QSQ and related algorithms as bottom-up, fixpoint evaluations. (Such meta-programming is common in functional and logic programming but yields novel results in the context of datalog.) Reference [Bry89] goes on to describe several top-down and bottom-up datalog evaluation techniques within the framework, proving their correctness and providing a basis for comparison.

A recent investigation [NRSU89] improves the performance of the magic sets in some cases. If the program and query satisfy certain conditions, then a technique called factoring can be used to replace some predicates by new predicates of lower arity. Other improvements are considered in [Sag90], where it is shown in particular that the advantage of one method over another may depend on the actual data, therefore stressing the need for techniques to estimate the size of *idb*'s (e.g., [LN90]).

Extensions of the datalog evaluation techniques to stratified datalog$^\neg$ programs (see Chapter 15) include [BPR87, Ros91, SI88, KT88].

Another important direction of research has been the parallel evaluation of datalog programs. Heuristics are described in [CW89b, GST90, Hul89, SL91, WS88, WO90].

A novel approach to answering datalog queries efficiently is developed in [DT92, DS93]. The focus is on cases in which the same query is asked repeatedly as the underlying *edb* is changing. The answer of the query (and additional scratch paper relations) is materialized against a given *edb* state, and first-order queries are used incrementally to maintain the materialized data as the underlying *edb* state is changed.

A number of prototype systems based on variants of datalog have been developed, incorporating some of the techniques mentioned in this chapter. They include DedGin [Vie87b, LV89], NAIL! [Ull85, MUV86, MNS$^+$87], LDL [NT89], ALGRES [CRG$^+$88], NU-Prolog [RSB$^+$87], GLUE-NAIL [DMP93], and CORAL [RSS92, RSSS93]. Descriptions of projects in this area can also be found in [Zan87], [RU94].

Exercises

Exercise 13.1 Recall the program RSG′ from Section 13.1. Exhibit an instance **I** such that on this input, $\delta^i_{rsg} \neq \emptyset$ for each $i > 0$.

Exercise 13.2 Recall the informal discussion of the two seminaive "versions" of the nonlinear ancestor program discussed in Section 13.1. Let P_1 denote the first of these, and P_2 the second. Show the following.

(a) For some input, P_2 can produce the same tuple more than once at some level beyond the first level.

(b) If P_2 produces the same tuple more than once, then each occurrence corresponds to a distinct proof tree (see Section 12.5) from the program and the input.

(c) P_1 can produce a given tuple twice, where the proof trees corresponding to the two occurrences are identical.

Exercise 13.3 Consider the basic seminaive algorithm (13.1.1).

(a) Verify that this algorithm terminates on all inputs.

(b) Show that for each $i \geq 0$ and each *idb* predicate S, after the i^{th} execution of the loop the value of variable S^i is equal to $T_P^i(\mathbf{I})(S)$ and the value of Δ_S^{i+1} is equal to $T_P^{i+1}(\mathbf{I})(S) - T_P^i(\mathbf{I})(S)$.

(c) Verify that this algorithm produces correct output on all inputs.

(d) Give an example input for which the same tuple is generated during different loops of the algorithm.

Exercise 13.4 Consider the improved seminaive algorithm (13.1.2).

(a) Verify that this algorithm terminates and produces correct output on all inputs.

(b) Give an example of a program P for which the improved seminaive algorithm produces fewer redundant tuples than the basic seminaive algorithm.

Exercise 13.5 Let P be a linear datalog program, and let P' be the set of rules associated with P by the improved seminaive algorithm. Suppose that the naive algorithm is performed using P' on some input $\mathbf{I}$. Does this yield $P(\mathbf{I})$? Why or why not? What if the basic seminaive algorithm is used?

Exercise 13.6 A set X of relevant facts for datalog query (P, q) and input $\mathbf{I}$ is *minimal* if (1) for each answer β of q there is a proof tree for β constructed from facts in X, and (2) X is minimal having this property. Informally describe an algorithm that produces a minimal set of relevant facts for a query (P, q) and input $\mathbf{I}$ and is polynomial time in the size of $\mathbf{I}$.

Exercise 13.7 [BR91] Suppose that program P includes the rule

$$\rho : S(x, y) \leftarrow S_1(x, z), S_2(z, y), S_3(u, v), S_4(v, w),$$

where S_3, S_4 are *edb* relations. Observe that the atoms $S_3(u, v)$ and $S_4(v, w)$ are not connected to the other atoms of the rule body or to the rule head. Furthermore, in an evaluation of P on input $\mathbf{I}$, this rule may contribute some tuple to S only if there is an assignment α for u, v, w such that $\{S_3(u, v), S_4(v, w)\}[\alpha] \subseteq \mathbf{I}$. Explain why it is typically more efficient to replace ρ with

$$\rho' : S(x, y) \leftarrow S_1(x, z), S_2(z, y)$$

if there is such an assignment and to delete ρ from P otherwise. Extend this to the case when S_3, S_4 are *idb* relations. State a general version of this heuristic improvement.

Exercise 13.8 Consider the adorned rule

$$R^{bf}(x, w) \leftarrow S_1^{bf}(x, y), S_2^{bf}(y, z), T_1^{ff}(u, v), T_2^{bf}(v, w).$$

Explain why it makes sense to view the second occurrence of v as bound.

Exercise 13.9 Consider the rule

$$R(x, y, y) \leftarrow S(y, z), T(z, x).$$

(a) Construct adorned versions of this rule for R^{ffb} and R^{fbb}.

(b) Suppose that in the QSQ algorithm a tuple $\langle b, c\rangle$ is placed into $input\_R^{fbb}$. Explain why this tuple should not be placed into the 0^{th} supplementary relation for the second adorned rule constructed in part (a).

(c) Exhibit an example analogous to part (b) based on the presence of a constant in the head of a rule rather than on repeated variables.

Exercise 13.10

(a) Complete the evaluation in Example 13.2.1.

(b) Use Algorithm 13.2.3 (QSQR) to evaluate that example.

★ **Exercise 13.11** In the QSQR algorithm, the procedure for processing subqueries of the form (R^γ, S) is called until no global variable is changed. Exhibit an example datalog query and input where the second cycle of calls to the subqueries (R^γ, S) generates new answer tuples.

♠ **Exercise 13.12** (a) Prove Theorem 13.2.2. (b) Prove that the QSQR algorithm is correct.

★ **Exercise 13.13** The *Iterative QSQ* (*QSQI*) algorithm uses the QSQ framework, but without recursion. Instead in each iteration it processes each rule body from left to right, using the values currently in the relations ans_R^γ when computing values for the supplementary relations. As with QSQR, the variables $input\_R^\gamma$ and ans_R^γ are global, and the variables for the supplementary relations are local. Iteration continues until there is no change to the global variables.

(a) Specify the QSQI algorithm more completely.

(b) Give an example where QSQI performs redundant work that QSQR does not.

Exercise 13.14 [BR91] Consider the following query based on a nonlinear variant of the same-generation program, called here the SGV query:

(a) $sgv(x, y) \leftarrow flat(x, y)$

(b) $sgv(x, y) \leftarrow up(x, z1), sgv(z1, z2), flat(z2, z3), sgv(z3, z4), down(z4, y)$
$query(y) \leftarrow sgv(a, y)$

Give the magic set transformation of this program and query.

Exercise 13.15 Give examples of how a query (P^m, q^m) resulting from magic set rewriting can produce nonrelevant and redundant facts.

♠ **Exercise 13.16**

(a) Give the general definition of the magic set rewriting technique.

(b) Prove Theorem 13.3.1.

Exercise 13.17 Compare the difficulties, in practical terms, of using the QSQ and magic set frameworks for evaluating datalog queries.

★ **Exercise 13.18** Let (P, q) denote the SGV query of Exercise 13.14. Let (P^m, q^m) denote the result of rewriting this program, using the (generalized supplementary) magic set transformation presented in this chapter. Under an earlier version, called here "original" magic, the rewritten form of (P, q) is (P^{om}, q^{om}):

(o-m1) $sgv^{bf}(x, y) \leftarrow input\_sgv^{bf}(x), flat(x, y)$

(o-m2) $sgv^{bf}(x, y) \leftarrow input\_sgv^{bf}(x), up(x, z1), sgv^{bf}(z1, z2), flat(z2, z3), sgv^{bf}(z3, z4), down(z4, y)$

(o-i2.2) $input\_sgv^{bf}(z1) \leftarrow input\_sgv^{bf}(x), up(x, z1)$

(o-i2.4) $input\_sgv^{bf}(z3) \leftarrow input\_sgv^{bf}(x), up(x, z1), sgv^{bf}(z1, z2), flat(z2, z3)$

(o-seed) $input\_sgv(a) \leftarrow$

(o-query) $query(y) \leftarrow sgv^{bf}(a, y)$

Intuitively, the original magic set transformation uses the relations $input\_R^{\gamma}$, but not supplementary relations.

(a) Verify that (P^{om}, q^{om}) is equivalent to (P, q).

(b) Compare the family of facts computed during the executions of (P^m, q^m) and (P^{om}, q^{om}).

(c) Give a specification for the original magic set transformation, applicable to any datalog query.

★ **Exercise 13.19** Consider the adorned rule

$$R^{bbf}(x, y, z) \leftarrow T_1^{bf}(x, s), T_2^{bf}(s, t), T_3^{bf}(y, u), T_4^{bf}(u, v), T_5^{bbf}(t, v, z).$$

A *sip graph* for this rule has as nodes all atoms of the rule and a special node *exit*, and edges (R, T_1), (T_1, T_2), (R, T_3), (T_3, T_4), (T_2, T_5), (T_4, T_5), $(T_5, exit)$. Describe a family of supplementary relations, based on this sip graph, that can be used in conjunction with the QSQ and magic set approaches. [Use one supplementary relation for each edge (corresponding to the output of the tail of the edge) and one supplementary relation for each node except for R (corresponding to the input to this node—in general, this will equal the join of the relations for the edges entering the node).] Explain how this may increase efficiency over the left-to-right approach used in this chapter. Generalize the construction. (The notion of sip graph and its use is a variation of [BR91].)

♠ **Exercise 13.20** [Ull89a] Specify an algorithm that replaces a program and query by an equivalent one, all of whose *idb* subgoals are rectified. What is the complexity of this algorithm?

♠ **Exercise 13.21**

(a) Provide a more detailed specification of the QSQ framework with annotations, and prove its correctness.

(b) [Ull89b, Ull89a] State formally the definitions needed for Theorem 13.4.1, and prove it.

Exercise 13.22 Write a program using counting that can be used to answer the RSG query presented at the beginning of Section 13.2.

(c-s1.1) $count\_sgv^{bf}(I, K, L, y) \leftarrow count\_input\_sgv^{bf}(I, K, L, x), flat(x, y)$

(c-s2.1) $count\_sup_1^2(I, K, L, z1) \leftarrow count\_input\_sgv^{bf}(I, K, L, x), up(x, z1)$

(c-s2.2) $count\_sup_2^2(I, K, L, z2) \leftarrow count\_sup_1^2(I, K, L, z1),$
$count\_sgv^{bf}(I + 1, 2K + 2, 5L + 2, z2)$

(c-s2.3) $count\_sup_3^2(I, K, L, z3) \leftarrow count\_sup_2^2(I, K, L, z2), flat(z2, z3)$

(c-s2.4) $count\_sup_4^2(I, K, L, z4) \leftarrow count\_sup_3^2(I, K, L, z3),$
$count\_sgv^{bf}(I + 1, 2K + 2, 5L + 4, z4),$

(c-s2.5) $count\_sgv^{bf}(I, K, L, y) \leftarrow count\_sup_4^2(I, K, L, z4), down(z4, y)$

(c-i2.2) $count\_input\_sgv^{bf}(I + 1, 2K + 2, 5L + 2, z1)$
$\leftarrow count\_sup_1^2(I, K, L, z1)$

(c-i2.4) $count\_input\_sgv^{bf}(I + 1, 2K + 2, 5L + 4, z3)$
$\leftarrow count\_sup_3^2(I, K, L, z3)$

(c-seed) $count\_input\_sgv^{bf}(1, 0, 0, a) \leftarrow$

(c-query) $query(y) \leftarrow count\_sgv^{bf}(1, 0, 0, y)$

Figure 13.9: Generalized counting transformation on SGV query

★ **Exercise 13.23** [BR91] This exercise illustrates a version of counting that is more general than that of Exercise 13.22. Indexed versions of predicates shall have three index coordinates (occurring leftmost) that hold:

(i) The level in the proof tree of the subgoal that a given rule is expanding.

(ii) An encoding of the rules used along the path from the root of the proof tree to the current subgoal. Suppose that there are k rules, numbered $(1), \ldots, (k)$. The index for the root node is 0 and, given index K, if rule number i is used next, then the next index is given by $kK + i$.

(iii) An encoding of the atom occurrence positions along the path from root to the current node. Assuming that l is the maximum number of *idb* atoms in any rule body, this index is encoded in a manner similar to item (ii).

A counting version of the SGV query of Exercise 13.14 is shown in Fig. 13.9. Verify that this is equivalent to the SGV query in the case where there are no cycles in *up* or *down*.

14 Recursion and Negation

Vittorio: *Let's combine recursion and negation.*
Riccardo: *That sounds hard to me.*
Sergio: *It's no problem, just add* fixpoint *to the calculus, or* while *to the algebra.*
Riccardo: *That sounds hard to me.*
Vittorio: *OK—how about datalog with negation?*
Riccardo: *That sounds hard to me.*
Alice: *Riccardo, you are recursively negative.*

The query languages considered so far were obtained by augmenting the conjunctive queries successively with disjunction, negation, and recursion. In this chapter, we consider languages that provide both negation and recursion. They allow us to ask queries such as, *"Which are the pairs of metro stops which are* **not** *connected?"*. This query is not expressible in relational calculus and algebra or in datalog.

The integration of recursion and negation is natural and yields highly expressive languages. We will see how it can be achieved in the three paradigms considered so far: algebraic, logic, and deductive. The algebraic language is an extension of the algebra with a looping construct and an assignment, in the style of traditional imperative programming languages. The logic language is an extension of the calculus in which recursion is provided by a fixpoint operator. The deductive language extends datalog with negation.

In this chapter, the semantics of datalog with negation is defined from a purely computational perspective that is in the spirit of the algebraic approach. More natural and widely accepted model-theoretic semantics, such as stratified and well-founded semantics, are presented in Chapter 15.

As we consider increasingly powerful languages, the complexity of query evaluation becomes a greater concern. We consider two flavors of the languages in each paradigm: the inflationary one, which guarantees termination in time polynomial in the size of the database; and the noninflationary one, which only guarantees that a polynomial amount of space is used.[1] In the last section of this chapter, we show that the polynomial-time-bounded languages defined in the different paradigms are equivalent. The set of queries they define is called the *fixpoint queries*. The polynomial-space-bounded languages are also equivalent, and the corresponding set of queries is called the *while* queries. In Chapter 17, we examine in more detail the expressiveness and complexity of the *fixpoint* and *while* queries. Note that, in particular, the polynomial time and space bounds on the complexity

[1] For comparison, it is shown in Chapter 17 that CALC requires only logarithmic space.

of such queries imply that there are queries that are not *fixpoint* or *while* queries. More powerful languages are considered in Chapter 18.

Before describing specific languages, we present an example that illustrates the principles underlying the two flavors of the languages.

EXAMPLE The following is based on a version of the well-known "game of life," which is used to model biological evolution. The game starts with a set of cells, some of which are alive and some dead; the alive ones are colored in blue or red. (One cell may have two colors.) Each cell has other cells as neighbors. Suppose that a binary relation *Neighbor* holds the neighbor relation (considered as a symmetric relation) and that the information about living cells and their color is held in a binary relation *Alive* (see Fig. 14.1). Suppose first that a cell can change status from dead to alive following this rule:

(α) A dead cell becomes alive if it has at least two neighbors that are alive and have the same color. It then takes the color of the "parents."

The evolution of a particular population for the *Neighbor* graph of Fig. 14.1(a) is given in Fig. 14.1(b). Observe that the sets of tuples keep increasing and that we reach a stable state. This is an example of inflationary iteration.

Now suppose that the evolution also obeys the second rule:

(β) A live cell dies if it has more than three live neighbors.

The evolution of the population with the two rules is given in Fig. 14.1(c). Observe that the number of tuples sometimes decreases and that the computation diverges. This is an example of noninflationary iteration.

All languages that we consider use a fixed set of relation schemas throughout the computation. At any point in the computation, intermediate results contain only constants from the input database or that are specified in the query. Suppose the relations used in the computation have arities $r_1, \ldots, r_k$, the input database contains n constants, and the query refers to c constants. Then the number of tuples in any intermediate result is bounded by $\sum_{i=1}^{k} (n + c)^{r_i}$, which is a polynomial in n. Thus such queries can be evaluated in polynomial space. As will be seen when the formal definitions are in place, this implies that each noninflationary iteration, and hence each noninflationary query, can be evaluated in polynomial space, whether or not it terminates. In contrast, the inflationary semantics ensures termination by requiring that a tuple can never be deleted once it has been inserted. Because there are only polynomially many tuples, each such program terminates in polynomial time.

To summarize, the *inflationary* languages use iteration based on an "inflation of tuples." In all three paradigms, inflationary queries can be evaluated in polynomial time, and the same expressive power is obtained. The *noninflationary* languages use noninflationary or destructive assignment inside of iterations. In all three paradigms, noninflationary queries can be evaluated in polynomial space, and again the same expressive power is

Neighbor

a	*e*
b	*e*
c	*e*
d	*e*

(a) Neighbor

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*
e	*blue*
e	*red*

Alive

a	*blue*	
b	*red*	
c	*blue*	. . .
d	*red*	
e	*blue*	
e	*red*	

(b) Inflationary evolution

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*
e	*blue*
e	*red*

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*
e	*blue*
e	*red*

Alive

a	*blue*	
b	*red*	. . .
c	*blue*	
d	*red*	

(c) Noninflationary evolution

Figure 14.1: Game of life

obtained. (We note, however, that it remains open whether the inflationary and the non-inflationary languages have equivalent expressive power; we discuss this issue later.)

14.1 Algebra + *While*

Relational algebra is essentially a procedural language. Of the query languages, it is the closest to traditional imperative programming languages. Chapters 4 and 5 described how it can be extended syntactically using assignment (:=) and composition (;) without increasing its expressive power. The extensions of the algebra with recursion are also consistent with

the imperative paradigm and incorporate a *while* construct, which calls for the iteration of a program segment. The resulting language comes in two flavors: inflationary and noninflationary. The two versions of the language differ in the semantics of the assignment statement. The noninflationary version was the one first defined historically, and we discuss it next. The resulting language is called the *while* language.

Noninflationary Semantics

Recall from Chapter 4 that assignment statements can be incorporated into the algebra using expressions of the form $R := E$, where E is an algebra expression and R a relational variable of the same sort as the result of E. (The difference from Chapter 4 is that it is no longer required that each successive assignment statement use a distinct, previously unused variable.) In the *while* language, the semantics of an assignment statement is as follows: The value of R becomes the result of evaluating the algebra expression E on the current state of the database. This is the usual destructive assignment in imperative programming languages, where the old value of a variable is overwritten.

While statements have the form

while change do
  begin
  ⟨*loop body*⟩
  end

There is no explicit termination condition. Instead a loop runs as long as the execution of the body causes some change to some relation (i.e., until a stable state is reached). At the end of this section, we consider the introduction of explicit terminating conditions and see that this does not affect the language in an essential manner.

Nesting of loops is permitted. A *while program* is a finite sequence of assignment or while statements. The program uses a finite set of relational variables of specified sorts, including the names of relations in the input database. Relational variables that are not in the input database are initialized to the empty relation. A designated relational variable holds the output to the program at the end of the computation. The *image* (or *value*) of program P on $\mathbf{I}$, denoted $P(\mathbf{I})$, is the value finally assigned to the designated variable if P terminates on $\mathbf{I}$; otherwise $P(\mathbf{I})$ is undefined.

Example 14.1.1 (Transitive Closure) Consider a binary relation $G[AB]$, specifying the edges of a graph. The following *while* program computes in $T[AB]$ the transitive closure of G.

$T := G$;
while change do
  begin
  $T := T \cup \pi_{AB}(\delta_{B\to C}(T) \bowtie \delta_{A\to C}(G))$;
  end

A computation ends when T becomes stable, which means that no new edges were added in the current iteration, so T now holds the transitive closure of G.

EXAMPLE 14.1.2 (Add-Remove) Consider again a binary relation G specifying the edges of a graph. Each loop of the following program

- removes from G all edges $\langle a, b\rangle$ if there is a path of length 2 from a to b, and
- inserts an edge $\langle a, b\rangle$ if there is a vertex not directly connected to a and b.

This is iterated while some change occurs. The result is placed into the binary relation T. In addition, the binary relation variables *ToAdd* and *ToRemove* are used as "scratch paper." For the sake of readability, we use the calculus with active domain semantics whenever this is easier to understand than the corresponding algebra expression.

$T := G$;
while change do
  begin
  $ToRemove := \{\langle x, y\rangle \mid \exists z(T(x, z) \wedge T(z, y))\}$;
  $ToAdd := \{\langle x, y\rangle \mid \exists z(\neg T(x, z) \wedge \neg T(z, x) \wedge \neg T(y, z) \wedge \neg T(z, y))\}$;
  $T := (T \cup ToAdd) - ToRemove$;
  end

In the *Transitive Closure* example, the transitive closure query always terminates. This is not the case for the *Add-Remove* query. (Try the graph $\{\langle a, a\rangle, \langle a, b\rangle, \langle b, a\rangle, \langle b, b\rangle\}$.) The halting problem for *while* programs is undecidable (i.e., there is no algorithm that, given a *while* program P, decides whether P halts on each input; see Exercise 14.2). Observe, however, that for a pair $(P, \mathbf{I})$, one can decide whether P halts on input $\mathbf{I}$ because, as argued earlier, *while* computations are in PSPACE.

Inflationary Semantics

We define next an inflationary version of the *while* language, denoted by *while*$^+$. The *while*$^+$ language differs with *while* in the semantics of the assignment statement. In particular, in *while*$^+$, assignment is cumulative rather than destructive: Execution of the statement assigning E to R results in *adding* the result of E to the old value of R. Thus no tuple is removed from any relation throughout the execution of the program. To distinguish the cumulative semantics from the destructive one, we use the notation $P \mathrel{+}= e$ for the cumulative semantics.

EXAMPLE 14.1.3 (Transitive Closure Revisited) Following is a *while*$^+$ program that computes the transitive closure of a graph represented by a binary relation $G[AB]$. The result is obtained in the variable $T[AB]$.

$T \mathrel{+}= G$;
while change do
  begin
  $T \mathrel{+}= \pi_{AB}(\delta_{B\to C}(T) \bowtie \delta_{A\to C}(G))$;
  end

This is almost exactly the same program as in the *while* language. The only difference is that because assignment is cumulative, it is not necessary to add the content of T to the result of the projection.

To conclude this section, we consider alternatives for the control condition of loops. Until now, we based termination on reaching a stable state. It is also common to use explicit terminating conditions, such as tests for emptiness of the form $E = \emptyset$, $E \neq \emptyset$, or $E \neq E'$, where E, E' are relational algebra expressions. The body of the loop is executed as long as the condition is satisfied. The following example shows how transitive closure is computed using explicit looping conditions.

Example 14.1.4 We use another relation schema $oldT$ also of sort AB.

$T \mathrel{+}= G;$
while $(T - oldT) \neq \emptyset$ *do*
  begin
  $oldT \mathrel{+}= T;$
  $T \mathrel{+}= \pi_{AB}(\delta_{B\to C}(T) \bowtie \delta_{A\to C}(G));$
  end

In the program, $oldT$ keeps track of the value of T resulting from the previous iteration of the loop. The computation ends when $oldT$ and T coincide, which means that no new edges were added in the current iteration, so T now holds the transitive closure of G.

It is easily shown that the use of such termination conditions does not modify the expressive power of *while*, and the use of conditions such as $E \neq E'$ does not modify the expressive power of *while*$^+$ (see Exercise 14.5).

In Section 14.4 we shall see that nesting of loops in *while* queries does not increase expressive power.

14.2 Calculus + Fixpoint

Just as in the case of the algebra, we provide inflationary and noninflationary extensions of the calculus with recursion. This could be done using assignment statements and while loops, as for the algebra. Indeed, we used calculus notation in Example 14.1.2 (*Add-Remove*). Instead we use an equivalent but more logic-oriented construct to augment the calculus. The construct, called a *fixpoint operator*, allows the iteration of calculus formulas up to a fixpoint. In effect, this allows defining relations inductively using calculus formulas. As with *while*, the fixpoint operator comes in a noninflationary and an inflationary flavor.

For the remainder of this chapter, as a notational convenience, we use active domain semantics for calculus queries. In addition, we often use a formula $\varphi(x_1, \ldots, x_n)$ as an abbreviation for the query $\{x_1, \ldots, x_n \mid \varphi(x_1, \ldots, x_n)\}$. These two simplifications do not affect the results developed.

Partial Fixpoints

The noninflationary version of the fixpoint operator is considered first. It is illustrated in the following example.

Example 14.2.1 (Transitive Closure Revisited) Consider again the transitive closure of a graph G. The relations J_n holding pairs of nodes at distance at most n can be defined inductively using the single formula

$$\varphi(T) = G(x, y) \ \vee \ T(x, y) \ \vee \ \exists z(T(x, z) \wedge G(z, y))$$

as follows:

$$J_0 = \emptyset;$$
$$J_n = \varphi(J_{n-1}), \quad n > 0.$$

Here $\varphi(J_{n-1})$ denotes the result of evaluating $\varphi(T)$ when the value of T is J_{n-1}. Note that, for each input G, the sequence $\{J_n\}_{n\geq 0}$ converges. That is, there exists some k for which $J_k = J_j$ for every $j > k$ (indeed, k is the diameter of the graph). Clearly, J_k holds the transitive closure of the graph. Thus the transitive closure of G can be defined as the limit of the foregoing sequence. Note that $J_k = \varphi(J_k)$, so J_k is also a *fixpoint* of $\varphi(T)$. The relation J_k thereby obtained is denoted by $\mu_T(\varphi(T))$. Then the transitive closure of G is defined by

$$\mu_T(G(x, y) \vee T(x, y) \vee \exists z(T(x, z) \wedge G(z, y))).$$

By definition, μ_T is an operator that produces a new relation (the fixpoint J_k) when applied to $\varphi(T)$. Note that, although T is used in $\varphi(T)$, T is not a database relation but rather a relation used to define inductively $\mu_T(\varphi(T))$ from the database, starting with $T = \emptyset$. T is said to be *bound* to μ_T. Indeed, μ_T is somewhat similar to a quantifier over relations. Note that the scope of the free variables of $\varphi(T)$ is restricted to $\varphi(T)$ by the operator μ_T.

In the preceding example, the limit of the sequence $\{J_n\}_{n\geq 0}$ happens to exist and is in fact the least fixpoint of φ. This is not always the case; the possibility of nontermination is illustrated next (and Exercise 14.4 considers cases in which a nonminimal fixpoint is reached).

Example 14.2.2 Consider

$$\varphi(T) = (x = 0 \wedge \neg T(0) \wedge \neg T(1)) \vee (x = 0 \wedge T(1)) \vee (x = 1 \wedge T(0)).$$

In this case the sequence $\{J_n\}_{n\geq 0}$ is $\emptyset$, $\{\langle 0\rangle\}$, $\{\langle 1\rangle\}$, $\{\langle 0\rangle\}$, ... (i.e., T flip-flops between zero and one). Thus the sequence does not converge, and $\mu_T(\varphi(T))$ is not defined. Situations in which μ is undefined correspond to nonterminating computations in the *while* language. The following nonterminating *while* program corresponds to $\mu_T(\varphi(T))$.

```
T := {⟨0⟩};
while change do
  begin
  T := {⟨0⟩, ⟨1⟩} − T;
  end
```

Because μ is only partially defined, it is called the *partial fixpoint operator*. We now define its syntax and semantics in more detail.

Partial Fixpoint Operator Let $\mathbf{R}$ be a database schema, and let $T[m]$ be a relation schema not in $\mathbf{R}$. Let $\mathbf{S}$ denote the schema $\mathbf{R} \cup \{T\}$. Let $\varphi(T)$ be a formula using T and relations in $\mathbf{R}$, with m free variables. Given an instance $\mathbf{I}$ over $\mathbf{R}$, $\mu_T(\varphi(T))$ denotes the relation that is the limit, *if it exists*, of the sequence $\{J_n\}_{n\geq 0}$ defined by

$$J_0 = \emptyset;$$
$$J_n = \varphi(J_{n-1}), \quad n > 0,$$

where $\varphi(J_{n-1})$ denotes the result of evaluating φ on the instance $\mathbf{J}_{n-1}$ over $\mathbf{S}$ whose restriction to $\mathbf{R}$ is $\mathbf{I}$ and $\mathbf{J}_{n-1}(T) = J_{n-1}$.

The expression $\mu_T(\varphi(T))$ denotes a new relation (if it is defined). In turn, it can be used in more complex formulas like any other relation. For example, $\mu_T(\varphi(T))(y, z)$ states that $\langle y, z\rangle$ is in $\mu_T(\varphi(T))$. If $\mu_T(\varphi(T))$ defines the transitive closure of G, the complement of the transitive closure is defined by

$$\{\langle x, y\rangle \mid \neg\, \mu_T(\varphi(T))(x, y)\}.$$

The extension of the calculus with μ is called *partial fixpoint logic*, denoted CALC+μ.

Partial Fixpoint Logic CALC+μ *formulas* are obtained by repeated applications of CALC operators ($\exists$, $\forall$, $\vee$, $\wedge$, $\neg$) and the partial fixpoint operator, starting from atoms. In particular, $\mu_T(\varphi(T))(e_1, \ldots, e_n)$, where T has arity n, $\varphi(T)$ has n free variables, and the e_i are variables or constants, is a formula. Its free variables are the variables in the set $\{e_1, \ldots, e_n\}$ [thus the scope of variables occurring inside $\varphi(T)$ consists of the subformula to which μ_T is applied]. Partial fixpoint operators can be nested. CALC+μ *queries* over a database schema $\mathbf{R}$ are expressions of the form

$$\{\langle e_1, \ldots, e_n\rangle \mid \xi\},$$

where ξ is a CALC+μ formula whose free variables are those occurring in $e_1, \ldots, e_n$. The formula ξ may use relation names in addition to those in $\mathbf{R}$; however, each occurrence P of such relation name must be bound to some partial fixpoint operator μ_P. The semantics of CALC+μ queries is defined as follows. First note that, given an instance $\mathbf{I}$ over $\mathbf{R}$ and a sentence σ in CALC+μ, there are three possibilities: σ is undefined on $\mathbf{I}$; σ is defined on $\mathbf{I}$

and is true; and σ is defined on $\mathbf{I}$ and is false. In particular, given an instance $\mathbf{I}$ over $\mathbf{R}$, the answer to the query

$$q = \{\langle e_1, \ldots, e_n\rangle \mid \xi\}$$

is undefined if the application of some μ in a subformula is undefined. Otherwise the answer to q is the n-ary relation consisting of all valuations ν of $e_1, \ldots, e_n$ for which $\xi(\nu(e_1), \ldots, \nu(e_n))$ is defined and true. The queries expressible in partial fixpoint logic are called the *partial fixpoint queries*.

Example 14.2.3 (Add-Remove Revisited) Consider again the query in Example 14.1.2. To express the query in CALC+μ, a difficulty arises: The *while* program initializes T to G before the while loop, whereas CALC+μ lacks the capability to do this directly. To distinguish the initialization step from the subsequent ones, we use a ternary relation Q and two distinct constants: 0 and 1. To indicate that the first step has been performed, we insert in Q the tuple $\langle 1, 1, 1\rangle$. The presence of $\langle 1, 1, 1\rangle$ in Q inhibits the repetition of the first step. Subsequently, an edge $\langle x, y\rangle$ is encoded in Q as $\langle x, y, 0\rangle$. The *while* program in Example 14.1.2 is equivalent to the CALC+μ query

$$\{\langle x, y\rangle \mid \mu_Q(\varphi(Q))(x, y, 0)\}$$

where

$$\begin{aligned}
&\varphi(Q) = \\
&\quad [\neg Q(1,1,1) \wedge [(G(x,y) \wedge z = 0) \vee (x = 1 \wedge y = 1 \wedge z = 1)]] \\
&\quad \vee \\
&\quad [Q(1,1,1) \wedge [(x = 1 \wedge y = 1 \wedge z = 1) \vee \\
&\qquad\qquad ((z = ((z = 0) \wedge Q(x, y, 0) \wedge \neg\exists w(Q(x, w, 0) \wedge Q(w, y, 0))) \vee \\
&\qquad\qquad ((z = ((z = 0) \wedge \exists w(\neg Q(x, w, 0) \wedge \neg Q(w, x, 0) \wedge \\
&\qquad\qquad\quad \neg Q(y, w, 0) \wedge \neg Q(w, y, 0)))]].
\end{aligned}$$

Clearly, this query is more awkward than its counterpart in *while*. The simulation highlights some peculiarities of computing with CALC+μ.

In Section 14.4 it is shown that the family of partial fixpoint queries is equivalent to the $while$ queries. In the preceding definition of $\mu_T(\varphi(T))$, the scope of all free variables in φ is defined by μ_T. For example, if T is binary in the following

$$\exists y(P(y) \wedge \mu_T(\varphi(T, x, y))(z, w)),$$

then $\varphi(T, x, y)$ has free variables x, y. According to the definition, y is *not* free in $\mu_T(\varphi(T, x, y))(z, w)$ (the free variables are z, w). Hence the quantifier $\exists y$ applies to the y in $P(y)$ alone and has no relation to the y in $\mu_T(\varphi(T, x, y))(z, w)$. To avoid confusion, it is preferable to use distinct variable names in such cases. For instance, the preceding

sentence can be rewritten as

$$\exists y(P(y) \wedge \mu_T(\varphi(T, x', y'))(z, w)).$$

A variant of the fixpoint operator can be developed that permits free variables under the fixpoint operator, but this does not increase the expressive power (see Exercise 14.11).

Simultaneous Induction

Consider the following use of nested partial fixpoint operators, where G, P, and Q are binary:

$$\mu_P(G(x, y) \wedge \mu_Q(\varphi(P, Q))(x, y)).$$

Here $\varphi(P, Q)$ involves both P and Q. This corresponds to a nested iteration. In each iteration i in the computation of $\{J_n\}_{n\geq 0}$ over P, the fixpoint $\mu_Q(\varphi(P, Q))$ is recomputed for the successive values J_i of P.

In contrast, we now consider a generalization of the partial fixpoint that permits simultaneous iteration over two or more relations. For example, let $\mathbf{R}$ be a database schema and $\varphi(P, Q)$ and $\psi(P, Q)$ be calculus formulas using P and Q not in $\mathbf{R}$, such that the arity of P (respectively Q) is the number of free variables in φ (ψ). On input $\mathbf{I}$ over $\mathbf{R}$, one can define inductively the sequence $\{\mathbf{J}_n\}_{n\geq 0}$ of relations over $\{P, Q\}$ as follows:

$$\begin{aligned}
\mathbf{J}_0(P) &= \emptyset \\
\mathbf{J}_0(Q) &= \emptyset \\
\mathbf{J}_n(P) &= \varphi(\mathbf{J}_{n-1}(P), \mathbf{J}_{n-1}(Q)) \\
\mathbf{J}_n(Q) &= \psi(\mathbf{J}_{n-1}(P), \mathbf{J}_{n-1}(Q)).
\end{aligned}$$

Such a mutually recursive definition of $\mathbf{J}_n(P)$ and $\mathbf{J}_n(Q)$ is referred to as *simultaneous induction*. If the sequence $\{\mathbf{J}_n(P), \mathbf{J}_n(Q)\}_{n\geq 0}$ converges, the limit is a fixpoint of the mapping on pairs of relations defined by $\varphi(P, Q)$ and $\psi(P, Q)$. This pair of values for P and Q is denoted by $\mu_{P,Q}(\varphi(P, Q), \psi(P, Q))$, and $\mu_{P,Q}$ is a *simultaneous induction partial fixpoint operator*. The value for P in $\mu_{P,Q}$ is denoted by $\mu_{P,Q}(\varphi(P, Q), \psi(P, Q))(P)$ and the value for Q by $\mu_{P,Q}(\varphi(P, Q), \psi(P, Q))(Q)$. Clearly, simultaneous induction definitions like the foregoing can be extended for any number of relations. Simultaneous induction can simplify certain queries, as shown next.

Example 14.2.4 (Add-Remove by Simultaneous Induction) Consider again the query *Add-Remove* in Example 14.2.3. One can simplify the query by introducing an auxiliary unary relation *Off*, which inhibits the transfer of G into T after the first step in a direct fashion. T and *Off* are defined in a mutually recursive fashion by φ_{Off} and φ_T, respectively:

$$\varphi_{Off}(x) = x = 1$$
$$\varphi_T(x, y) = [\neg Off(1) \wedge G(x, y)]$$
$$\vee\ [Off(1) \wedge \neg\exists z(T(x, z) \wedge T(z, y)) \wedge$$
$$(T(x, y) \vee \exists z(\neg T(x, z) \wedge \neg T(z, x) \wedge \neg T(y, z) \wedge \neg T(z, y))].$$

The *Add-Remove* query can now be written as

$$\{\langle x, y\rangle \mid \mu_{Off,T}(\varphi_{Off}(Off, T), \varphi_T(Off, T))(T)(x, y)\}.$$

It turns out that using simultaneous induction instead of regular fixpoint operators does not provide additional power. For example, a CALC+μ formula equivalent to the query in Example 14.2.4 is the one shown in Example 14.2.3. More generally, we have the following:

Lemma 14.2.5 For some n, let $\varphi_i(R_1, \ldots, R_n)$ be CALC formulas, i in $[1..n]$, such that $\mu_{R_1,\ldots,R_n}(\varphi_1(R_1, \ldots, R_n), \ldots, \varphi_n(R_1, \ldots, R_n))$ is a correct formula. Then for each $i \in [1, n]$ there exist CALC formulas $\varphi_i'(Q)$ and tuples $\vec{e}_i$ of variables or constants such that for each i,

$$\mu_{R_1,\ldots,R_n}(\varphi_1(R_1, \ldots, R_n), \ldots, \varphi_n(R_1, \ldots, R_n))(R_i) \equiv \mu_Q(\varphi_i'(Q))(\vec{e}_i).$$

Crux We illustrate the construction with reference to the query of Example 14.2.4. Instead of using two relations *Off* and T, we use a ternary relation Q that encodes both *Off* and T. The extra coordinate is used to distinguish between tuples in T and tuples in *Off*. A tuple $\langle x\rangle$ in *Off* is encoded as a tuple $\langle x, 1, 1\rangle$ in Q. A tuple $\langle x, y\rangle$ in T is encoded as a tuple $\langle x, y, 0\rangle$ in Q. The final result is obtained by selecting from Q the tuples where the third coordinate is 0 and projecting the result on the first two coordinates. ■

Note that the use of the tuples $\vec{e}_i$ allows one to perform appropriate selections and projections on $\mu_Q(\varphi_i'(Q))$ necessary for decoding. These selections and projections are essential and cannot be avoided (see Exercise 14.17c).

Inflationary Fixpoint

The nonconvergence in some cases of the sequence $\{J_n\}_{n\geq 0}$ in the semantics of the partial fixpoint operator is similar to nonterminating computations in the *while* language with noninflationary semantics. The semantics of the partial fixpoint operator μ is essentially noninflationary because in the inductive definition of J_n, each step is a destructive assignment. As with *while*, we can make the semantics inflationary by having the assignment at each step of the induction be cumulative. This yields an inflationary version of μ, denoted by μ^+ and called the *inflationary fixpoint operator*, which is defined for all formulas and databases to which it is applied.

Inflationary Fixpoint Operators and Logic The definition of $\mu_T^+(\varphi(T))$ is identical to that of the partial fixpoint operator except that the sequence $\{J_n\}_{n\geq 0}$ is defined as follows:

$$J_0 = \emptyset;$$
$$J_n = J_{n-1} \cup \varphi(J_{n-1}), \quad n > 0.$$

This definition ensures that the sequence $\{J_n\}_{n\geq 0}$ is increasing: $J_{i-1} \subseteq J_i$ for each $i > 0$. Because for each instance there are finitely many tuples that can be added, the sequence converges in all cases.

Adding μ^+ instead of μ to CALC yields *inflationary fixpoint logic*, denoted by CALC+μ^+. Note that inflationary fixpoint queries are always defined.

The set of queries expressible by inflationary fixpoint logic is called the *fixpoint queries*. The fixpoint queries were historically defined first among the inflationary languages in the algebraic, logic, and deductive paradigms. Therefore the class of queries expressible in inflationary languages in the three paradigms has come to be referred to as the fixpoint queries.

As a simple example, the transitive closure of a graph G is defined by the following CALC+μ^+ query:

$$\{\langle x, y\rangle \mid \mu_T^+(G(x, y) \vee \exists z(T(x, z) \wedge G(z, y))(x, y)\}.$$

Recall that datalog as presented in Chapter 12 uses an inflationary operator and yields the minimal fixpoint of a set of rules. One may also be tempted to assume that an inflationary simultaneous induction of the form $\mu_{P,Q}^+(\varphi(P, Q), \psi(P, Q))$ is equivalent to a system of equational definitions of the form

$$P = \varphi(P, Q)$$
$$Q = \psi(P, Q)$$

and that it computes the unique minimal fixpoint for P and Q. However, one should be careful because the result of the inflationary fixpoint computation is only one of the possible fixpoints. As illustrated in the following example, this may not be minimal or the "naturally" expected fixpoint. (There may not exist a unique minimal fixpoint; see Exercise 14.4.)

Example 14.2.6 Consider the equation

$$T(x, y) = G(x, y) \vee T(x, y) \vee \exists z(T(x, z) \wedge G(z, y))$$
$$CT(x, y) = \neg T(x, y).$$

One is tempted to believe that the fixpoint of these two equations yields the complement of transitive closure. However, with the inflationary semantics

$$
\begin{aligned}
\mathbf{J}_0(T) &= \emptyset \\
\mathbf{J}_0(CT) &= \emptyset \\
\mathbf{J}_n(T) &= \mathbf{J}_{n-1}(T) \cup \{\langle x, y\rangle \mid G(x, y) \vee \mathbf{J}_{n-1}(T)(x, y) \\
&\qquad \vee \exists z(\mathbf{J}_{n-1}(T)(x, z) \wedge G(z, y))\} \\
\mathbf{J}_n(CT) &= \mathbf{J}_{n-1}(CT) \cup \{\langle x, y\rangle \mid \neg\mathbf{J}_{n-1}(T)(x, y)\}
\end{aligned}
$$

leads to saturating CT at the first iteration.

Positive and Monotone Formulas

Making the fixpoint operator inflationary by definition is not the only way to guarantee polynomial-time termination of the fixpoint iteration. An alternative approach is to restrict the formulas $\varphi(T)$ so that convergence of the sequence $\{J_n\}_{n\geq 0}$ associated with $\mu_T(\varphi(T))$ is guaranteed. One such restriction is monotonicity. Recall that a query q is *monotone* if for each **I, J**, $\mathbf{I} \subseteq \mathbf{J}$ then $q(\mathbf{I}) \subseteq q(\mathbf{J})$. One can again show that for such formulas, a least fixpoint always exists and that it is obtained after a finite (but unbounded) number of stages of inductive applications of the formula.

Unfortunately, monotonicity is an undecidable property for CALC. One can also restrict the application of fixpoint to positive formulas. This was historically the first track that was followed and presents the advantage that positiveness is a decidable (syntactic) property. It is done by requiring that T occur only *positively* in $\varphi(T)$ (i.e., under an even number of negations in the syntax tree of the formula). All formulas thereby obtained are monotone, and so $\mu_T(\varphi(T))$ is always defined (see Exercise 14.10).

It can be shown that the approach of inflationary fixpoint and the two approaches based on fixpoint of positive or monotone formulas are equivalent (i.e., the sets of queries expressed are identical; see Exercise 14.10).

Fixpoint Operators and Circumscription

In some sense, the fixpoint operators act as quantifiers on relational variables. This is somewhat similar to the well-known technique of *circumscription* studied in artificial intelligence. Suppose $\psi(T)$ is a calculus sentence (i.e., no free variables) that uses T in addition to relations from a database schema $\mathbf{R}$. The circumscription of $\psi(T)$ with respect to T, denoted here by $circ_T(\psi(T))$, can be thought of as an operator defining a new relation, starting from the database. More precisely, let $\mathbf{I}$ be an instance over $\mathbf{R}$. Then $circ_T(\psi(T))$ denotes the relation containing all tuples belonging to every relation T such that (1) $\psi(T)$ holds for $\mathbf{I}$, and (2) T is minimal under set inclusion[2] with this property. Consider now a fixpoint query. As stated earlier, fixpoint queries can be expressed using just fixpoint operators μ_T applied to formulas positive in T (i.e., T always appears in φ under an even number of negations). We claim that $\mu_T(\varphi(T)) = circ_T(\varphi'(T))$, where $\varphi'(T)$ is a sentence

[2] Other kinds of minimality have also been considered.

obtained from $\varphi(T)$ as follows:

$$\varphi'(T) = \forall x_1, \ldots \forall x_n(\varphi(T, x_1, \ldots, x_n) \rightarrow T(x_1, \ldots, x_n)),$$

where the arity of T is n. To see this, it is sufficient to note that $\mu_T(\varphi(T))$ is the unique minimal T satisfying $\varphi'(T)$. This uses the monotonicity of $\varphi(T)$ with respect to T, which follows from the fact that $\varphi(T)$ is positive in T (see Exercise 14.10). Although computing with circumscription is generally intractable, the fixpoint operator on positive formulas can always be evaluated in polynomial time. Thus the fixpoint operator can be viewed as a tractable restriction of circumscription.

14.3 Datalog with Negation

Datalog provides recursion but no negation. It defines only monotonic queries. Viewed from the standpoint of the deductive paradigm, datalog provides a form of *monotonic reasoning*. Adding negation to datalog rules permits the specification of nonmonotonic queries and hence of *nonmonotonic reasoning*.

Adding negation to datalog rules requires defining semantics for negative facts. This can be done in many ways. The different definitions depend to some extent on whether datalog is viewed in the deductive framework or simply as a specification formalism like any other query language. In this chapter, we examine the latter point of view. Then datalog with negation can essentially be viewed as a subset of the *while* or *fixpoint* queries and can be treated similarly. This is not necessarily appropriate in the deductive framework. For instance, the basic assumptions in the reasoning process may require that once a fact is assumed false at some point in the inferencing process, it should not be proven true at a later point. This idea lies at the core of stratified and well-founded semantics, two of the most widely accepted in the deductive framework. The deductive point of view is considered in depth in Chapter 15.

The semantics given here for datalog with negation follows the semantics given in Chapter 12 for datalog, but does not correspond directly to the semantics for nonrecursive datalog$^\neg$ given in Chapter 5. The semantics in Chapter 5 is inspired by the stratified semantics but can be simulated by (either of) the semantics presented in this chapter.

As in the previous section, we consider both inflationary and noninflationary versions of datalog with negation.

Inflationary Semantics

The inflationary language allows negations in bodies of rules and is denoted by *datalog*$^\neg$. Like datalog, its rules are used to infer a set of facts. Once a fact is inferred, it is never removed from the set of true facts. This yields the inflationary character of the language.

Example 14.3.1 We present a datalog$^\neg$ program with input a graph in binary relation G. The program computes the relation $closer(x, y, x', y')$ defined as follows:

closer(x, y, x', y') means that the distance $d(x, y)$ from x to y in G is smaller than the distance $d(x', y')$ from x' to y' [$d(x, y)$ is infinite if there is no path from x to y].

$$\begin{aligned} T(x, y) &\leftarrow G(x, y) \\ T(x, y) &\leftarrow T(x, z), G(z, y) \\ \textit{closer}(x, y, x', y') &\leftarrow T(x, y), \neg T(x', y') \end{aligned}$$

The program is evaluated as follows. The rules are fired simultaneously with all applicable valuations. At each such firing, some facts are inferred. This is repeated until no new facts can be inferred. A negative fact such as $\neg T(x', y')$ is true if $T(x', y')$ has not been inferred so far. This does not preclude $T(x', y')$ from being inferred at a later firing of the rules. One firing of the rules is called a stage in the evaluation of the program. In the preceding program, the transitive closure of G is computed in T. Consider the consecutive stages in the evaluation of the program. Note that if the fact $T(x, y)$ is inferred at stage n, then $d(x, y) = n$. So if $T(x', y')$ has not been inferred yet, this means that the distance between x and y is less than that between x' and y'. Thus if $T(x, y)$ and $\neg T(x', y')$ hold at some stage n, then $d(x, y) \leq n$ and $d(x', y') > n$ and *closer*(x, y, x', y') is inferred.

The formal syntax and semantics of datalog$^\neg$ are straightforward extensions of those for datalog. A datalog$^\neg$ *rule* is an expression of the form

$$A \leftarrow L_1, \ldots, L_n,$$

where A is an atom and each L_i is either an atom B_i (in which case it is called *positive*) or a negated atom $\neg B_i$ (in which case it is called *negative*). (In this chapter we use an active domain semantics for evaluating datalog$^\neg$ and so do not require that the rules be range restricted; see Exercise 14.13.)

A datalog$^\neg$ *program* is a nonempty finite set of datalog$^\neg$ rules. As for datalog programs, $sch(P)$ denotes the database schema consisting of all relations involved in the program P; the relations occurring in heads of rules are the *idb* relations of P, and the others are the *edb* relations of P.

The semantics of datalog$^\neg$ that we present in this chapter is an extension of the fixpoint semantics of datalog. Let $\mathbf{K}$ be an instance over $sch(P)$. Recall that an (active domain) *instantiation* of a rule $A \leftarrow L_1, \ldots, L_n$ is a rule $\nu(A) \leftarrow \nu(L_1), \ldots, \nu(L_n)$, where ν is a valuation that maps each variable into $adom(P, \mathbf{K})$. A fact A' is an *immediate consequence* for $\mathbf{K}$ and P if $A' \in \mathbf{K}(R)$ for some *edb* relation R, or $A' \leftarrow L'_1, \ldots, L'_n$ is an instantiation of a rule in P and each positive L'_i is a fact in $\mathbf{K}$, and for each negative $L'_i = \neg A'_i$, $A'_i \notin \mathbf{K}$. The *immediate consequence operator* of P, denoted Γ_P, is now defined as follows. For each $\mathbf{K}$ over $sch(P)$,

$$\Gamma_P(\mathbf{K}) = \mathbf{K} \cup \{A \mid A \text{ is an immediate consequence for } \mathbf{K} \text{ and } P\}.$$

Given an instance $\mathbf{I}$ over $edb(P)$, one can compute $\Gamma_P(\mathbf{I})$, $\Gamma_P^2(\mathbf{I})$, $\Gamma_P^3(\mathbf{I})$, etc. As suggested in Example 14.3.1, each application of Γ_P is called a *stage* in the evaluation. From the

definition of Γ_P, it follows that

$$\Gamma_P(\mathbf{I}) \subseteq \Gamma_P^2(\mathbf{I}) \subseteq \Gamma_P^3(\mathbf{I}) \subseteq \ldots.$$

As for datalog, the sequence reaches a fixpoint, denoted $\Gamma_P^\infty(\mathbf{I})$, after a finite number of steps. The restriction of this to the *idb* relations (or some subset thereof) is called the *image* (or *answer*) of P on $\mathbf{I}$.

An important difference with datalog is that $\Gamma_P^\infty(\mathbf{I})$ is no longer guaranteed to be a minimal model of P containing $\mathbf{I}$, as illustrated next.

Example 14.3.2 Let P be the program

$$R(0) \leftarrow Q(0), \neg R(1)$$
$$R(1) \leftarrow Q(0), \neg R(0).$$

Let $\mathbf{I} = \{Q(0)\}$. Then $P(\mathbf{I}) = \{Q(0), R(0), R(1)\}$. Although $P(\mathbf{I})$ is a model of P, it is not minimal. The minimal models containing $\mathbf{I}$ are $\{Q(0), R(0)\}$ and $\{Q(0), R(1)\}$.

As discussed in Chapter 12, the operational semantics of datalog based on the immediate consequence operator is equivalent to the natural semantics based on minimal models. As shown in the preceding example, there may not be a unique minimal model for a datalog$^\neg$ program, and the semantics given for datalog$^\neg$ may not yield any of the minimal models. The development of a natural model-theoretic semantics for datalog$^\neg$ thus calls for selecting a natural model from among several possible candidates. Inevitably, such choices are open to debate; Chapter 15 presents several alternatives.

Noninflationary Semantics

The language datalog$^\neg$ has inflationary semantics because the set of facts inferred through the consecutive firings of the rules is increasing. To obtain a noninflationary variant, there are several possibilities. One could keep the syntax of datalog$^\neg$ but make the semantics noninflationary by retaining, at each stage, only the newly inferred facts (see Exercise 14.16). Another possibility is to allow explicit retraction of a previously inferred fact. Syntactically, this can be done using negations in heads of rules, interpreted as deletions of facts. We adopt this solution here, in part because it brings our language closer to some practical languages that use so-called (production) rules in the sense of expert and active database systems. The resulting language is denoted by datalog$^{\neg\neg}$, to indicate that negations are allowed in both heads and bodies of rules.

Example 14.3.3 (Add-Remove Visited Again) The following datalog$^{\neg\neg}$ program computes in T the *Add-Remove* query of Example 14.1.2, given as input a graph G.

$$
\begin{aligned}
T(x, y) &\leftarrow G(x, y), \neg \mathit{off}(1) \\
\mathit{off}(1) &\leftarrow \\
\neg T(x, y) &\leftarrow T(x, z), T(z, y), \mathit{off}(1) \\
T(x, y) &\leftarrow \neg T(x, z), \neg T(z, x), \neg T(y, z), \neg T(z, y), \mathit{off}(1)
\end{aligned}
$$

Relation *off* is used to inhibit the first rule (initializing T to G) after the first step.

The immediate consequence operator Γ_P and semantics of a datalog$^{\neg\neg}$ program are analogous to those for datalog$^{\neg}$, with the following important proviso. If a negative literal $\neg A$ is inferred, the fact A is removed, unless A is also inferred in the same firing of the rules. This gives priority to inference of positive over negative facts and is somewhat arbitrary. Other possibilities are as follows: (1) Give priority to negative facts; (2) interpret the simultaneous inference of A and $\neg A$ as a "no-op" (i.e., including A in the new instance only if it is there in the old one); and (3) interpret the simultaneous inference of A and $\neg A$ as a contradiction that makes the result undefined. The chosen semantics has the advantage over possibility (3) that the semantics is always defined. In any case, the choice of semantics is not crucial: They yield equivalent languages (see Exercise 14.15).

With the semantics chosen previously, termination is no longer guaranteed. For instance, the program

$$
\begin{aligned}
T(0) &\leftarrow T(1) \\
\neg T(1) &\leftarrow T(1) \\
T(1) &\leftarrow T(0) \\
\neg T(0) &\leftarrow T(0)
\end{aligned}
$$

never terminates on input $T(0)$. The value of T flip-flops between $\{\langle 0\rangle\}$ and $\{\langle 1\rangle\}$, so no fixpoint is reached.

Datalog$^{\neg\neg}$ and Datalog$^{\neg}$ as Fragments of CALC+μ and CALC+μ^+

Consider datalog$^{\neg\neg}$. It can be viewed as a subset of CALC+μ in the following manner. Suppose that P is a datalog$^{\neg\neg}$ program. The *idb* relations defined by rules can alternately be defined by simultaneous induction using formulas that correspond to the rules. Each firing of the rules corresponds to one step in the simultaneous inductive definition. For instance, the simultaneous induction definition corresponding to the program in Example 14.3.3 is the one in Example 14.2.4. Because simultaneous induction can be simulated in CALC+μ (see Lemma 14.2.5), datalog$^{\neg\neg}$ can be simulated in CALC+μ. Moreover, notice that only a single application of the fixpoint operator is used in the simulation. Similar remarks apply to datalog$^{\neg}$ and CALC+μ^+. Furthermore, in the inflationary case it is easy to see that the formula can be chosen to be *existential* (i.e., its prenex normal form[3] uses only existential

[3] A CALC formula in prenex normal form is a formula $Q_1 x_1 \ldots Q_k x_k \varphi$ where Q_i, $1 \leq i \leq k$ are quantifiers and φ is quantifier free.

quantifiers). The same can be shown in the noninflationary case, although the proof is more subtle. In summary (see Exercise 14.18), the following applies:

Lemma 14.3.4 Each datalog$^{\neg\neg}$ (datalog$^{\neg}$) query is equivalent to a CALC+μ (CALC+μ^+) query of the form

$$\{\vec{x} \mid \mu_T^{(+)}(\varphi(T))(\vec{t})\},$$

where

(a) φ is an existential CALC formula, and

(b) $\vec{t}$ is a tuple of variables or constants of appropriate arity and $\vec{x}$ is the tuple of distinct free variables in $\vec{t}$.

The Rule Algebra

The examples of datalog$^{\neg}$ programs shown in this chapter make it clear that the semantics of such programs is not always easy to understand. There is a simple mechanism that facilitates the specification by the user of various customized semantics. This is done by means of the *rule algebra*, which allows specification of an order of firing of the rules as well as firing up to a fixpoint in an inflationary or noninflationary manner. For the inflationary version RA^+ of the rule algebra, the base expressions are individual datalog$^{\neg}$ rules; the semantics associated with a rule is to apply its immediate consequence operator once in a cumulative fashion. Union ($\cup$) can be used to specify simultaneous application of a pair of rules or more complex programs. The expression $P; Q$ specifies the composition of P and Q; its semantics is to execute P once and then Q once. Inflationary iteration of program P is called for by $(P)^+$. The noninflationary version of the rule algebra, denoted RA, starts with datalog$^{\neg}$ rules, but now with a noninflationary, destructive semantics, as defined in Exercise 14.16. Union and composition are generalized in the natural fashion, and the noninflationary iterator, denoted *, is used.

Example 14.3.5 Let P be the set of rules

$$T(x, y) \leftarrow G(x, y)$$
$$T(x, y) \leftarrow T(x, z), G(z, y)$$

and let Q consist of the rule

$$CT(x, y) \leftarrow \neg T(x, y).$$

The RA^+ program $(P)^+; Q$ computes in CT the complement of the transitive closure of G.

It follows easily from the results of Section 14.4 that RA^+ is equivalent to datalog$^{\neg}$, and RA is equivalent to noninflationary datalog$^{\neg}$ and hence to datalog$^{\neg\neg}$ (Exercise 14.23). Thus an RA^+ program can be compiled into a (possibly much more complicated) datalog$^{\neg}$

program. For instance, the RA^+ program in Example 14.3.5 is equivalent to the datalog$^\neg$ program in Example 14.4.2. The advantage of the rule algebra is the ease of expressing various semantics. In particular, RA^+ can be used easily to specify the stratified and well-founded semantics for datalog$^\neg$ introduced in Chapter 15.

14.4 Equivalence

The previous sections introduced inflationary and noninflationary recursive languages with negation in the algebraic, logic, and deductive paradigms. This section shows that the inflationary languages in the three paradigms, *while*$^+$, CALC+μ^+, and datalog$^\neg$, are equivalent and that the same holds for the noninflationary languages *while*, CALC+μ, and datalog$^{\neg\neg}$. This yields two classes of queries that are central in the theory of query languages: the *fixpoint* queries (expressed by the inflationary languages) and the *while* queries (expressed by the noninflationary languages). This is summarized in Fig. 14.2, at the end of the chapter.

We begin with the equivalence of the inflationary languages because it is the more difficult to show. The equivalence of CALC+μ^+ and *while*$^+$ is easy because the languages have similar capabilities: Program composition in *while*$^+$ corresponds closely to formula composition in CALC+μ^+, and the *while change* loop of *while*$^+$ is close to the inflationary fixpoint operator of CALC+μ^+. More difficult and surprising is the equivalence of these languages with datalog$^\neg$, because this much simpler language has no explicit constructs for program composition or nested recursion.

Lemma 14.4.1 CALC+μ^+ and *while*$^+$ are equivalent.

Proof We consider first the simulation of CALC+μ^+ queries by *while*$^+$. Let $\{\langle x_1,\ldots,x_m\rangle \mid \xi(x_1,\ldots,x_m)\}$ be a CALC+μ^+ query over an input database with schema $\mathbf{R}$. It suffices to show that there exists a *while*$^+$ program P_ξ that defines the same result as $\xi(x_1, \ldots, x_m)$ in some m-ary relation R_ξ. The proof is by induction on the depth of nesting of the fixpoint operator in ξ, denoted $d(\xi)$. If $d(\xi) = 0$ (i.e., ξ does not contain a fixpoint operator), then ξ is in CALC and P_ξ is

$$R_\xi \ += \ E_\xi,$$

where E_ξ is the relational algebra expression corresponding to ξ. Now suppose the statement is true for formulas with depth of nesting of the fixpoint operator less than $d (d > 0)$. Let ξ be a formula with $d(\xi) = d$.

If $\xi = \mu_Q(\varphi(Q))(f_1, \ldots, f_k)$, then P_ξ is

$Q \mathrel{+}= \emptyset$;
while change do
  begin
  E_φ;
  $Q \mathrel{+}= R_\varphi$
  end;
  $R_\xi \mathrel{+}= \pi(\sigma(Q))$,

where $\pi(\sigma(Q))$ denotes the selection and projection corresponding to $f_1, \ldots, f_k$.

Suppose now that ξ is obtained by first-order operations from k formulas $\xi_1, \ldots, \xi_k$, each having μ^+ as root. Let $E_\xi(R_{\xi_1}, \ldots, R_{\xi_k})$ be the relational algebra expression corresponding to ξ, where each subformula $\xi_i = \mu_Q(\varphi(Q))(e^i_1, \ldots, e^i_{n_i})$ is replaced by R_{ξ_i}. For each i, let P_{ξ_i} be a program that produces the value of $\mu_Q(\varphi(Q))(e^i_1, \ldots, e^i_{n_i})$ and places it into R_{ξ_i}. Then P_ξ is

$$P_{\xi_1}; \ldots; \; P_{\xi_k};$$
$$R_\xi \mathrel{+}= E_\xi(R_{\xi_1}, \ldots, R_{\xi_k}).$$

This completes the induction and the proof that CALC+μ^+ can be simulated by *while*$^+$. The converse simulation is similar (Exercise 14.20). ■

We now turn to the equivalence of CALC+μ^+ and datalog$^\neg$. Lemma 14.3.4 yields the subsumption of datalog$^\neg$ by CALC+μ^+. For the other direction, we simulate CALC+μ^+ queries using datalog$^\neg$. This simulation presents two main difficulties.

The first involves delaying the firing of a rule until after the completion of a fixpoint by another set of rules. Intuitively, this is hard because checking that the fixpoint has been reached involves checking the *nonexistence* rather than the existence of some valuation, and datalog$^\neg$ is more naturally geared toward checking the *existence* of valuations. The solution to this difficulty is illustrated in the following example.

Example 14.4.2 The following datalog$^\neg$ program computes the complement of the transitive closure of a graph G. The example illustrates the technique used to delay the firing of a rule (computing the complement) until the fixpoint of a set of rules (computing the transitive closure) has been reached (i.e., until the application of the transitivity rule yields no new tuples). To monitor this, the relations *old-T*, *old-T-except-final* are used. *old-T* follows the computation of T but is one step behind it. The relation *old-T-except-final* is identical to *old-T* but the rule defining it includes a clause that prevents it from firing when T has reached its last iteration. Thus *old-T* and *old-T-except-final* differ only in the iteration after the transitive closure T reaches its final value. In the subsequent iteration, the program recognizes that the fixpoint has been reached and fires the rule computing the complement in relation CT. The program is

$$\begin{aligned}
&T(x, y) && \leftarrow G(x, y)\\
&T(x, y) && \leftarrow G(x, z), T(z, y)\\
&\textit{old-T}(x, y) && \leftarrow T(x, y)\\
&\textit{old-T-except-final}(x, y) && \leftarrow T(x, y), T(x', z'), T(z', y'), \neg T(x', y')\\
&CT(x, y) && \leftarrow \neg T(x, y), \textit{old-T}(x', y'),\\
& && \qquad \neg \textit{old-T-except-final}(x', y')
\end{aligned}$$

(It is assumed that G is not empty; see Exercise 14.3.)

The second difficulty concerns keeping track of iterations in the computation of a fixpoint. Given a formula $\mu^+_T(\varphi(T))$, the simulation of φ itself may involve numerous relations other than T, whose behavior may be "sabotaged" by an overly zealous application of iteration of the immediate consequence operator. To overcome this, we separate the internal computation of φ from the external iteration over T, as illustrated in the following example.

Example 14.4.3 Let G be a binary relation schema. Consider the CALC+μ^+ query $\mu^+_{good}(\phi(good))(x)$, where

$$\phi = \forall y\ (G(y, x) \rightarrow good(y)).$$

Note that the query computes the set of nodes in G that are not reachable from a cycle (in other words, the nodes such that the length of paths leading to them is bounded). One application of $\varphi(good)$ is achieved by the datalog$^\neg$ program P:

$$\begin{aligned} bad(x) &\leftarrow G(y, x), \neg good(y) \\ delay &\leftarrow \\ good(x) &\leftarrow delay, \neg bad(x) \end{aligned}$$

Simply iterating P does not yield the desired result. Intuitively, the relations *delay* and *bad*, which are used as "scratch paper" in the computation of a single iteration of μ^+, cannot be reinitialized and so cannot be reused to perform the computation of subsequent iterations.

To surmount this problem, we essentially create a version of P for each iteration of $\varphi(good)$. The versions are distinguished by using "timestamps." The nodes themselves serve as timestamps. The timestamps marking iteration i are the values newly introduced in relation *good* at iteration $i - 1$. Relations *delay* and *delay-stamped* are used to delay the derivation of new tuples in *good* until *bad* and *bad-stamped* (respectively) have been computed in the current iteration. The process continues until no new values are introduced in an iteration. The full program is the union of the three rules given earlier, which perform the first iteration, and the following rules, which perform the iteration with timestamp t:

$$\begin{aligned} bad\text{-}stamped(x, t) &\leftarrow G(y, x), \neg good(y), good(t) \\ delay\text{-}stamped(t) &\leftarrow good(t) \\ good(x) &\leftarrow delay\text{-}stamped(t), \neg bad\text{-}stamped(x, t). \end{aligned}$$

We now embark on the formal demonstration that datalog$^\neg$ can simulate CALC+μ^+. We first introduce some notation relating to the timestamping of a program in the simulation. Let $m \geq 1$. For each relation schema Q, let $\overline{Q}$ be a new relational schema with $\text{arity}(\overline{Q}) = arity(Q) + m$. If $(\neg)Q(e_1, \ldots, e_n)$ is a literal and $\vec{z}$ an m-tuple of distinct variables, then $(\neg)Q(e_1, \ldots, e_n)[\vec{z}]$ denotes the literal $(\neg)\overline{Q}(e_1, \ldots, e_n, z_1, \ldots, z_m)$. For each program P and tuple $\vec{z}$, $P[\vec{z}]$ denotes the program obtained from P by replacing each literal A by $A[\vec{z}]$. Let P be a program and $B_1, \ldots, B_q$ a list of literals. Then P // $B_1, \ldots, B_q$ is the program obtained by appending $B_1, \ldots, B_q$ to the bodies of all rules in P.

To illustrate the previous notation, consider the program P consisting of the following two rules:

$$S(x, y) \leftarrow R(x, y)$$
$$S(x, y) \leftarrow R(x, z), S(z, y).$$

Then $P[z] \,//\, \neg T(x, w, y)$ is

$$\overline{S}(x, y, z) \leftarrow \overline{R}(x, y, z), \neg T(x, w, y)$$
$$\overline{S}(x, y, z) \leftarrow \overline{R}(x, z, z), \overline{S}(z, y, z), \neg T(x, w, y).$$

Lemma 14.4.4 CALC+μ^+ and datalog$^\neg$ are equivalent.

Proof As seen in Lemma 14.3.4, datalog$^\neg$ is essentially a fragment of CALC+μ^+, so we just need to show the simulation of CALC+μ^+ by datalog$^\neg$. The proof is by structural induction on the CALC+μ^+ formula. The core of the proof involves a control mechanism that delays firing certain rules until other rules have been evaluated. Therefore the induction hypothesis involves the capability to simulate the CALC+μ^+ formula using a datalog$^\neg$ program as well as to produce concomitantly a predicate that only becomes true when the simulation has been completed. More precisely, we will prove by induction the following: For each CALC+μ^+ formula φ over a database schema $\mathbf{R}$, there exists a datalog$^\neg$ program $prog(\varphi)$ whose *edb* relations are the relations in $\mathbf{R}$, whose *idb* relations include $result_\varphi$ with arity equal to the number of free variables in φ and a 0-ary relation $done_\varphi$ such that for every instance $\mathbf{I}$ over $\mathbf{R}$,

(i) $[prog(\varphi)(\mathbf{I})](result_\varphi) = \varphi(\mathbf{I})$, and

(ii) the 0-ary predicate $done_\varphi$ becomes true at the last stage in the evaluation of $prog(\varphi)$ on $\mathbf{I}$.

We will assume, without loss of generality, that no variable of φ occurs free and bound, or bound to more than one quantifier, that φ contains no $\forall$ or $\vee$, and that the initial query has the form $\{x_1, \ldots, x_n \mid \xi\}$, where $x_1, \ldots, x_n$ are distinct variables. Note that the last assumption implies that (i) establishes the desired result.

Suppose now that φ is an atom $\mathrm{R}(\vec{e})$. Let $\vec{x}$ be the tuple of distinct variables occurring in $\vec{e}$. Then $prog(\varphi)$ consists of the rules

$$done_\varphi \quad \leftarrow$$
$$result_\varphi(\vec{x}) \leftarrow R(\vec{e}).$$

There are four cases to consider for the induction step.

1. $\varphi = \alpha \wedge \beta$. Without loss of generality, we assume that the *idb* relations of $prog(\alpha)$ and $prog(\beta)$ are disjoint. Thus there is no interference between $prog(\alpha)$ and $prog(\beta)$. Let $\vec{x}$ and $\vec{y}$ be the tuples of distinct free variables of α and β, respectively, and let $\vec{z}$ be the tuple of distinct free variables occurring in $\vec{x}$ or $\vec{y}$.

Then $prog(\varphi)$ consists of the following rules:

$$\begin{aligned}
&prog(\alpha)\\
&prog(\beta)\\
&result_\varphi(\vec{z}) \leftarrow done_\alpha, done_\beta, result_\alpha(\vec{x}), result_\beta(\vec{y})\\
&done_\varphi \quad \leftarrow done_\alpha, done_\beta.
\end{aligned}$$

2. $\varphi = \exists\, x(\psi)$. Let $\vec{y}$ be the tuple of distinct free variables of ψ, and let $\vec{z}$ be the tuple obtained from $\vec{y}$ by removing the variable x. Then $prog(\varphi)$ consists of the rules

$$\begin{aligned}
&prog(\psi)\\
&result_\varphi(\vec{z}) \leftarrow done_\psi, result_\psi(\vec{y})\\
&done_\varphi \quad \leftarrow done_\psi.
\end{aligned}$$

3. $\varphi = \neg(\psi)$. Let $\vec{x}$ be the tuple of distinct free variables occurring in ψ. Then $prog(\varphi)$ consists of

$$\begin{aligned}
&prog(\psi)\\
&result_\varphi(\vec{x}) \leftarrow done_\psi, \neg result_\psi(\vec{x})\\
&done_\varphi \quad \leftarrow done_\psi.
\end{aligned}$$

4. $\varphi = \mu_S(\psi(S))(\vec{e})$. This case is the most involved, because it requires keeping track of the iterations in the computation of the fixpoint as well as bookkeeping to control the value of the special predicate $done_\varphi$. Intuitively, each iteration is marked by timestamps. The current timestamps consist of the tuples newly inserted in the previous iteration. The program $prog(\varphi)$ uses the following new auxiliary relations:
   - Relation $fixpoint_\varphi$ contains $\mu_S(\psi(S))$ at the end of the computation, and $result_\varphi$ contains $\mu_S(\psi(S))(\vec{e})$.
   - Relation run_φ contains the timestamps.
   - Relation $used_\varphi$ contains the timestamps introduced in the previous stages of the iteration. The active timestamps are in $run_\varphi - used_\varphi$.
   - Relation $not\text{-}final_\varphi$ is used to detect the final iteration (i.e., the iteration that adds no new tuples to $fixpoint_\varphi$). The presence of a timestamp in $used_\varphi - not\text{-}final_\varphi$ indicates that the final iteration has been completed.
   - Relations $delay_\varphi$ and $not\text{-}empty_\varphi$ are used for timing and to detect an empty result.

In the following, $\vec{y}$ and $\vec{t}$ are tuples of distinct variables with the same arity as S. We first have particular rules to perform the first iteration and to handle the special case of an empty result:

$$
\begin{aligned}
&\mathit{prog}(\psi)\\
&\mathit{fixpoint}_\varphi(\vec{y}) \leftarrow \mathit{result}_\psi(\vec{y}), \mathit{done}_\psi\\
&\mathit{delay}_\varphi \leftarrow \mathit{done}_\psi\\
&\mathit{not\text{-}empty}_\varphi \leftarrow \mathit{result}_\psi(\vec{y})\\
&\mathit{done}_\varphi \leftarrow \mathit{delay}_\varphi, \neg \mathit{not\text{-}empty}_\varphi.
\end{aligned}
$$

The remainder of the program contains the following rules:

- Stamping of the database and starting an iteration: For each R in ψ different from S and a tuple $\vec{x}$ of distinct variables with same arity as R,

$$
\begin{aligned}
\overline{R}(\vec{x}, \vec{t}) &\leftarrow R(\vec{x}), \mathit{fixpoint}_\varphi(\vec{t})\\
\mathit{run}_\varphi(\vec{t}) &\leftarrow \mathit{fixpoint}_\varphi(\vec{t})\\
\overline{S}(\vec{y}, \vec{t}) &\leftarrow \mathit{fixpoint}_\varphi(\vec{y}), \mathit{fixpoint}_\varphi(\vec{t}).
\end{aligned}
$$

- Timestamped iteration:

$$\mathit{prog}(\psi)[\vec{t}]//\mathit{run}_\varphi(\vec{t}), \neg \mathit{used}_\varphi(\vec{t})$$

- Maintain $\mathit{fixpoint}_\varphi$, $\mathit{not\text{-}last}_\varphi$, and used_φ:

$$
\begin{aligned}
\mathit{fixpoint}_\varphi(\vec{y}) &\leftarrow \overline{\mathit{done}}_\psi(\vec{t}), \overline{\mathit{result}}_\psi(\vec{y}, \vec{t}), \neg \mathit{used}_\varphi(\vec{t})\\
\mathit{not\text{-}final}_\varphi(\vec{t}) &\leftarrow \overline{\mathit{done}}_\psi(\vec{t}), \overline{\mathit{result}}_\psi(\vec{y}, \vec{t}), \neg \mathit{fixpoint}_\varphi(\vec{y})\\
\mathit{used}_\varphi(\vec{t}) &\leftarrow \overline{\mathit{done}}_\psi(\vec{t})
\end{aligned}
$$

- Produce the result and detect termination:

$$\mathit{result}_\varphi(\vec{z}) \leftarrow \mathit{fixpoint}_\varphi(\vec{e})$$

  where $\vec{z}$ is the tuple of distinct variables in $\vec{e}$,

$$\mathit{done}_\varphi \leftarrow \mathit{used}_\varphi(\vec{t}), \neg \mathit{not\text{-}final}_\varphi(\vec{t}).$$

It is easily verified by inspection that $\mathit{prog}(\varphi)$ satisfies (i) and (ii) under the induction hypothesis for cases (1) through (3). To see that (i) and (ii) hold in case (4), we carefully consider the stages in the evaluation of prog_φ. Let $\mathbf{I}$ be an instance over the relations in ψ other than S; let $J_0 = \emptyset$ be over S; and let $J_i = J_{i-1} \cup \psi(J_{i-1})$ for each $i > 0$. Then $\mu_S(\psi(S))(\mathbf{I}) = J_n$ for some n such that $J_n = J_{n-1}$. The program prog_φ simulates the consecutive iterations of this process. The first iteration is simulated using prog_ψ directly, whereas the subsequent iterations are simulated by prog_ψ timestamped with the tuples added at the previous iteration. (We omit consideration of the case in which the fixpoint is $\emptyset$; this is taken care of by the rules involving delay_φ and $\mathit{not\text{-}empty}_\varphi$.)

We focus on the stages in the evaluation of $prog_\varphi$ corresponding to the end of the simulation of each iteration of ψ. The stage in which the simulation of the first iteration is completed immediately follows the stage in which $done_\psi$ becomes true. The subsequent iterations are completed immediately following the stages in which

$$\exists \vec{t}(\overline{done}_\psi(\vec{t}) \wedge \neg used_\varphi(\vec{t}))$$

becomes true. Thus let k_1 be the stage in which $done_\psi$ becomes true, and let k_i $(2 < i \leq n)$ be the successive stages in which

$$\exists \vec{t}(\overline{done}_\psi(\vec{t}) \wedge \neg used_\varphi(\vec{t}))$$

is true. First note that

- at stage k_1

$$\{\vec{y} \mid result_\psi(\vec{y})\} = \psi(J_0);$$

- at stage $k_1 + 1$

$$fixpoint_\varphi = J_1.$$

For $i > 1$ it can be shown by induction on i that

- at stage k_i $(i \leq n)$

$$\{\vec{t} \mid \overline{done}_\psi(\vec{t}) \wedge \neg used_\varphi(\vec{t})\} = \psi(J_{i-2}) - J_{i-2} = J_{i-1} - J_{i-2}$$
$$\{\vec{y} \mid \overline{done}_\psi(\vec{t}) \wedge \overline{result}_\psi(\vec{y}, \vec{t}) \wedge \neg used_\varphi(\vec{t})\} = \psi(J_{i-1});$$
$$\{\vec{t} \mid \overline{done}_\psi(\vec{t}) \wedge \overline{result}_\psi(\vec{y}, \vec{t}) \wedge \neg fixpoint_\varphi(\vec{y})\} = \psi(J_{i-1}) - J_{i-1} = J_i - J_{i-1};$$

- at stage $k_i + 1$ $(i < n)$

$$fixpoint_\varphi = J_{i-1} \cup \psi(J_{i-1}) = J_i,$$
$$used_\varphi = not\text{-}last_\varphi = \overline{done}_\psi = J_{i-1};$$

- at stage $k_i + 2$ $(i < n)$

$$\{\vec{t} \mid run_\varphi(\vec{t}) \wedge \neg used_\varphi(\vec{t})\} = J_i - J_{i-1},$$
$$\{\vec{x} \mid \overline{R}(\vec{x}, \vec{t}) \wedge run_\varphi(\vec{t}) \wedge \neg used_\varphi(\vec{t})\} = \mathbf{I}(R),$$
$$\{\vec{x} \mid \overline{S}(\vec{x}, \vec{t}) \wedge run_\varphi(\vec{t}) \wedge \neg used_\varphi(\vec{t})\} = J_i.$$

Finally, at stage $k_n + 1$

$$used_\varphi = J_{n-1},$$

$$\textit{not-last}_\varphi = J_{n-2},$$
$$\textit{fixpoint}_\varphi = J_n = \mu_S(\psi(S))(\mathbf{I}),$$

and at stage $k_n + 2$

$$\textit{result}_\varphi = \mu_S(\psi(S))(\vec{z})(\mathbf{I}),$$
$$\textit{done}_\varphi = \mathbf{true}.$$

Thus (i) and (ii) hold for $\textit{prog}_\varphi$ in case (4), which concludes the induction. ■

Lemmas 14.4.1 and 14.4.4 now yield the following:

THEOREM 14.4.5 *while*$^+$, CALC+μ^+, and datalog$^\neg$ are equivalent.

The set of queries expressible in *while*$^+$, CALC+μ^+, and datalog$^\neg$ is called the *fixpoint queries*. An analogous equivalence result can be proven for the noninflationary languages *while*, CALC+μ, and datalog$^{\neg\neg}$. The proof of the equivalence of CALC+μ and datalog$^{\neg\neg}$ is easier than in the inflationary case because the ability to perform deletions in datalog$^{\neg\neg}$ facilitates the task of simulating explicit control (see Exercise 14.21). Thus we can prove the following:

THEOREM 14.4.6 *while*, CALC+μ, and datalog$^{\neg\neg}$ are equivalent.

The set of queries expressible in *while*, CALC+μ, and datalog$^{\neg\neg}$ is called the *while queries*. We will look at the *fixpoint* queries and the *while* queries from a complexity and expressiveness standpoint in Chapter 17. Although the spirit of our discussion in this chapter suggested that *fixpoint* and *while* are distinct classes of queries, this is far from obvious. In fact, the question remains open: As shown in Chapter 17, *fixpoint* and *while* are equivalent iff PTIME = PSPACE (Theorem 17.4.3).

The equivalences among languages discussed in this chapter are summarized in Fig. 14.2.

Normal Forms

The two equivalence theorems just presented have interesting consequences for the underlying extensions of datalog and logic. First they show that these languages are closed under composition and complementation. For instance, if two mappings f, g, respectively, from a schema S to a schema S' and from S' to a schema S'' are expressible in datalog$^{\neg(\neg)}$, then $f \circ g$ and $\neg f$ are also expressible in datalog$^{\neg(\neg)}$. Analogous results are true for CALC+$\mu^{(+)}$.

A more dramatic consequence concerns the nesting of recursion in the calculus and algebra. Consider first CALC+μ^+. By the equivalence theorems, this is equivalent to datalog$^\neg$, which, in turn (by Lemma 14.3.4), is essentially a fragment of CALC+μ^+. This yields a normal form for CALC+μ^+ queries and implies that a single application of

Languages		*Class of queries*
inflationary	*while*$^+$ CALC $+\mu^+$ *datalog*$^\neg$	*fixpoint*
noninflationary	*while* CALC $+\mu$ *datalog*$^{\neg\neg}$	*while*

Figure 14.2: Summary of language equivalence results

the inflationary fixpoint operator is all that is needed. Similar remarks apply to CALC+μ queries. In summary, the following applies:

THEOREM 14.4.7 Each CALC+$\mu^{(+)}$ query is equivalent to a CALC+$\mu^{(+)}$ query of the form

$$\{\, \vec{x} \mid \mu_T^{(+)}(\varphi(T))(\vec{t})\},$$

where φ is an existential CALC formula.

Analogous normal forms can be shown for *while*$^{(+)}$ (Exercise 14.22) and for $RA^{(+)}$ (Exercise 14.24).

14.5 Recursion in Practical Languages

To date, there are numerous prototypes (but no commercial product) that provide query and update languages with recursion. Many of these languages provide semantics for recursion in the spirit of the procedural semantics described in this chapter. Prototypes implementing the deductive paradigm are discussed in Chapter 15.

SQL 2-3 (a norm provided by ISO/ANSII) allows **select** statements that define a table used recursively in the **from** and **where** clauses. Such recursion is also allowed in Starburst. The semantics of the recursion is inflationary, although noninflationary semantics can be achieved using deletion. An extension of SQL 2-3 is ESQL (Extended SQL). To illustrate the flavor of the syntax (which is typical for this category of languages), the following is an ESQL program defining a table SPARTS (subparts), the transitive closure of the table PARTS. This is done using a view creation mechanism.

create view *SPARTS* **as**
select *
from *PARTS*
union

select *P1.PART, P2.COMPONENT*
from *SPARTS P1, PARTS P2*
where *P1.COMPONENT* = *P2.PART*;

This is in the spirit of CALC+μ^+. With deletion, one can simulate CALC+μ. The system Postgres also provides similar iteration up to a fixpoint in its query language POSTQUEL.

A form of recursion closer to *while* and *while*$^+$ is provided by SQL embedded in full programming languages, such as C+SQL, which allows SQL statements coupled with C programs. The recursion is provided by while loops in the host language.

The recursion provided by datalog$^\neg$ and datalog$^{\neg\neg}$ is close in spirit to *production-rule* systems. Speaking loosely, a production rule has the form

if ⟨condition⟩ then ⟨action⟩.

Production rules permit the specification of database updates, whereas deductive rules usually support only database queries (with some notable exceptions). Note that the deletion in datalog$^{\neg\neg}$ can be viewed as providing an update capability. The production-rule approach has been studied widely in connection with expert systems in artificial intelligence; OPS5 is a well-known system that uses this approach.

A feature similar to recursive rules is found in the emerging field of *active* databases. In active databases, the rule condition is often broken into two pieces; one piece, called the *trigger*, is usually closely tied to the database (e.g., based on insertions to or deletions from relations) and can be implemented deep in the system.

In active database systems, rules are recursively fired when conditions become true in the database. Speaking in broad terms, the noninflationary languages studied in this chapter can be viewed as an abstraction of this behavior. For example, the database language RDL1 is close in spirit to the language datalog$^{\neg\neg}$. (See also Chapter 22 for a discussion of active databases.)

The language Graphlog, a visual language for queries on graphs developed at the University of Toronto, emphasizes queries involving paths and provides recursion specified using regular expressions that describe the shape of desired paths.

Bibliographic Notes

The *while* language was first introduced as RQ in [CH82] and as LE in [Cha81a]. The other noninflationary languages, CALC+μ and datalog$^{\neg\neg}$, were defined in [AV91a]. The equivalence of the noninflationary languages was also shown there.

The fixpoint languages have a long history. Logics with fixpoints have been considered by logicians in the general case where infinite structures (corresponding to infinite database instances) are allowed [Mos74]. In the finite case, which is relevant in this book, the fixpoint queries were first defined using the partial fixpoint operator μ_T applied only to formulas positive in T [CH82]. The language allowing applications of μ_T to formulas monotonic, but not necessarily positive, in T was further studied in [Gur84]. An interesting difference between unrestricted and finite models arises here: Every CALC formula monotone in some predicate R is equivalent for unrestricted structures to some CALC formula positive in R (Lyndon's lemma), whereas this is not the case for finite structures [AG87]. Monotonicity is undecidable for both cases [Gur84].

The languages (1) with fixpoint over positive formulas, (2) with fixpoint over monotone formulas, and (3) with inflationary fixpoint over arbitrary formulas were shown equivalent in [GS86]. As a side-effect, it was shown in [GS86] that the nesting of μ (or μ^+) provides no additional power. This fact had been proven earlier for the first language in [Imm86]. Moreover, a new alternative proof of the sufficiency of a single application of the fixpoint in CALC+μ^+ is provided in [Lei90]. The simultaneous induction lemma (Lemma 14.2.5) was also proven in [GS86], extending an analogous result of [Mos74] for infinite structures. Of the other inflationary languages, *while*$^+$ was defined in [AV90] and datalog$^\neg$ with fixpoint semantics was first defined in [AV88c, KP88].

The equivalence of datalog$^\neg$ with CALC+μ^+ and *while*$^+$ was shown in [AV91a]. The relationship between the *while* and *fixpoint* queries was investigated in [AV91b], where it was shown that they are equivalent iff PTIME = PSPACE. The issues of complexity and expressivity of *fixpoint* and *while* queries will be considered in detail in Chapter 17.

The rule algebra for logic programs was introduced in [IN88].

The game of life is described in detail in [Gar70]. The normal forms discussed in this chapter can be viewed as variations of well-known folk theorems, described in [Har80].

SQL 2-3 is described in an ISO/ANSII norm [57391, 69392]). Starburst is presented in [HCL$^+$90]. ESQL (Extended SQL) is described in [GV92]. The example ESQL program in Section 14.5 is from [GV92]. The query language of Postgres, POSTQUEL, is presented in [SR86]. OPS5 is described in [For81].

The area of *active* databases is the subject of numerous works, including [Mor83, Coh89, KDM88, SJGP90, MD89, WF90, HJ91a]. Early work on database triggers includes [Esw76, BC79]. The language RDL1 is presented in [dMS88].

The visual graph language Graphlog, developed at the University of Toronto, is described in [CM90, CM93a, CM93b].

Exercises

Exercise 14.1 (Game of life) Consider the two rules informally described in Example 14.1.

(a) Express the corresponding queries in datalog$^{\neg(\neg)}$, *while*$^{(+)}$, and CALC+$\mu^{(+)}$.

(b) Find an input for which a vertex keeps changing color forever under the second rule.

Exercise 14.2 Prove that the termination problem for a *while* program is undecidable (i.e., that it is undecidable, given a *while* query, whether it terminates on all inputs). *Hint:* Use a reduction of the containment problem for algebra queries.

Exercise 14.3 Recall the datalog$^{\neg\neg}$ program of Example 14.4.2.

(a) After how many stages does the program complete for an input graph of diameter n?

(b) Modify the program so that it also handles the case of empty graphs.

(c) Modify the program so that it terminates in order of $\log(n)$ stages for an input graph of diameter n.

Exercise 14.4 Recall the definition of $\mu_T(\varphi(T))$.

(a) Exhibit a formula φ such that $\varphi(T)$ has a unique minimal fixpoint on all inputs, and $\mu_T(\varphi(T))$ terminates on all inputs but does not evaluate to the minimal fixpoint on any of them.

(b) Exhibit a formula φ such that $\mu_T(\varphi(T))$ terminates on all inputs but φ does not have a unique minimal fixpoint on any input.

Exercise 14.5

(a) Give a *while* program with explicit looping condition for the query in Example 14.1.2.

(b) Prove that $while^{(+)}$ with looping conditions of the form $E = \emptyset$, $E \neq \emptyset$, $E = E'$, and $E \neq E'$, where E, E' are algebra expressions, is equivalent to $while^{(+)}$ with the *change* conditions.

Exercise 14.6 Consider the problem of finding, given two graphs G, G' over the same vertex set, the minimum set X of vertexes satisfying the following conditions: (1) For each vertex v, if all vertexes v' such that there is a G-edge from v' to v are in X, then v is in X; and (2) the analogue for G'-edges. Exhibit a *while* program and a *fixpoint* query that compute this set.

Exercise 14.7 Recall the CALC+μ^+ query of Example 14.4.3.

(a) Run the query on the input graph G:
$\{\langle a, b\rangle, \langle c, b\rangle, \langle b, d\rangle, \langle d, e\rangle, \langle e, f\rangle, \langle f, g\rangle, \langle g, d\rangle, \langle e, h\rangle, \langle i, j\rangle, \langle j, h\rangle\}$.

(b) Exhibit a $while^+$ program that computes *good*.

(c) Write a program in your favorite conventional programming language (e.g., C or LISP) that computes the good vertexes of a graph G. Compare it with the database queries developed in this chapter.

(d) Show that a vertex a is *good* if there is no path from a vertex belonging to a cycle to a. Using this as a starting point, propose an alternative algorithm for computing the good vertexes. Is your algorithm expressible in *while*? In *fixpoint*?

★ **Exercise 14.8** Suppose that the input consists of a graph G together with a successor relation on the vertexes of G [i.e., a binary relation *succ* such that (1) each element has exactly one successor, except for one that has none; and (2) each element in the binary relation G occurs in *succ*].

(a) Give a *fixpoint* query that tests whether the input satisfies (1) and (2).

(b) Sketch a *while* program computing the set of pairs $\langle a, b\rangle$ such that the shortest path from a to b is a prime number.

(c) Do (b) using a $while^+$ query.

Exercise 14.9 (Simultaneous induction) Prove Lemma 14.2.5.

♠ **Exercise 14.10** (Fixpoint over positive formulas) Let $\varphi(T)$ be a formula positive in T (i.e., each occurrence of T is under an even number of negations in the syntax tree of φ). Let $\mathbf{R}$ be the set of relations other than T occurring in $\varphi(T)$.

(a) Show that $\varphi(T)$ is monotonic in T. That is, for all instances $\mathbf{I}$ and $\mathbf{J}$ over $\mathbf{R} \cup \{T\}$ such that $\mathbf{I}(\mathbf{R}) = \mathbf{J}(\mathbf{R})$ and $\mathbf{I}(T) \subseteq \mathbf{J}(T)$,

$$\varphi(\mathbf{I}) \subseteq \varphi(\mathbf{J}).$$

(b) Show that $\mu_T(\varphi(T))$ is defined on every input instance.

(c) [GS86] Show that the family of CALC+μ queries with fixpoints only over positive formulas is equivalent to the CALC+μ^+ queries.

★ **Exercise 14.11** Suppose CALC+μ^+ is modified so that free variables are allowed under fixpoint operators. More precisely, let

$$\varphi(T, x_1, \ldots, x_n, y_1, \ldots, y_m)$$

be a formula where T has arity n and the x_i and y_j are free in φ. Then

$$\mu_{T,x_1,\ldots,x_n}(\varphi(T, x_1, \ldots, x_n, y_1, \ldots, y_m))(e_1, \ldots, e_n)$$

is a correct formula, whose free variables are the y_j and those occurring among the e_i. The fixpoint is defined with respect to a given valuation of the y_j. For instance,

$$\exists z \exists w (P(z) \wedge \mu_{T,x,y}(\varphi(T, x, y, z))(u, w))$$

is a well-formed formula. Give a precise definition of the semantics for queries using this operator. Show that this extension does not yield increased expressive power over CALC+μ^+. Do the same for CALC+μ.

Exercise 14.12 Let G be a graph. Give a *fixpoint* query in each of the three paradigms that computes the pairs of vertexes such that the shortest path between them is of even length.

Exercise 14.13 Let datalog$_{rr}^{\neg(\neg)}$ denote the family of datalog$^{\neg(\neg)}$ programs that are *range restricted*, in the sense that for each rule r and each variable x occurring in r, x occurs in a positive literal in the body of r. Prove that datalog$_{rr}^{\neg} \equiv$ datalog$^{\neg}$ and datalog$_{rr}^{\neg\neg} \equiv$ datalog$^{\neg\neg}$.

Exercise 14.14 Show that negations in bodies of rules are redundant in datalog$^{\neg\neg}$ (i.e., for each datalog$^{\neg\neg}$ program P there exists an equivalent datalog$^{\neg\neg}$ program Q that uses no negations in bodies of rules). *Hint:* Maintain the complement of each relation R in a new relation R', using deletions.

♠ **Exercise 14.15** Consider the following semantics for negations in heads of datalog$^{\neg\neg}$ rules:

(α) the semantics giving priority to positive over negative facts inferred simultaneously (adopted in this chapter),

(β) the semantics giving priority to negative over positive facts inferred simultaneously,

(γ) the semantics in which simultaneous inference of A and $\neg A$ leads to a "no-op" (i.e., including A in the new instance only if it is there in the old one), and

(δ) the semantics prohibiting the simultaneous inference of a fact and its negation by making the result undefined in such circumstances.

For a datalog$^{\neg\neg}$ program P, let P_ξ, denote the program P with semantics $\xi \in \{\alpha, \beta, \gamma, \delta\}$.

(a) Give an example of a program P for which P_α, P_β, P_γ, and P_δ define distinct queries.

(b) Show that it is undecidable, for a given program P, whether P_δ never simultaneously infers a positive fact and its negation for any input.

(c) Let datalog$_\xi^{\neg\neg}$ denote the family of queries P_ξ for $\xi \in \{\alpha, \beta, \gamma\}$. Prove that datalog$_\alpha^{\neg\neg} \equiv$ datalog$_\beta^{\neg\neg} \equiv$ datalog$_\gamma^{\neg\neg}$.

(d) Give a syntactic condition on datalog$^{\neg\neg}$ programs such that under the δ semantics they never simultaneously infer a positve fact and its negation, and such that the resulting query language is equivalent to datalog$_\alpha^{\neg\neg}$.

Exercise 14.16 (Noninflationary datalog$^\neg$) The semantics of datalog$^\neg$ can be made noninflationary by defining the immediate consequence operator to be destructive in the sense that only the newly inferred facts are kept after each firing of the rules. Show that, with this semantics, datalog$^\neg$ is equivalent to datalog$^{\neg\neg}$.

★ **Exercise 14.17** (Multiple versus single carriers)

(a) Consider a datalog$^\neg$ program P producing the answer to a query in an *idb* relation S. Prove that there exists a program Q with the same *edb* relations as P and just one idb relation T such that, for each *edb* instance $\mathbf{I}$,

$$[P(\mathbf{I})](S) = \pi(\sigma([Q(\mathbf{I})](T))),$$

where σ denotes a selection and π a projection.

(b) Show that the projection π and selection σ in part (a) are indispensable. *Hint:* Suppose there is a datalog$^\neg$ program with a single *edb* relation computing the complement of transitive closure of a graph. Reach a contradiction by showing in this case that connectivity of a graph is expressible in relational calculus. (It is shown in Chapter 17 that connectivity is not expressible in the calculus.)

(c) Show that the projection and selection used in Lemma 14.2.5 are also indispensable.

★ **Exercise 14.18**

(a) Prove Lemma 14.3.4 for the inflationary case.

(b) Prove Lemma 14.3.4 for the noninflationary case. *Hint:* For datalog$^{\neg\neg}$, the straightforward simulation yields a formula $\mu_T(\varphi(T))(\vec{x})$, where φ may contain negations over existential quantifiers to simulate the semantics of deletions in heads of rules of the datalog$^{\neg\neg}$ program. Use instead the noninflationary version of datalog$^\neg$ described in Exercise 14.16.

Exercise 14.19 Prove that the simulation in Example 14.4.3 works.

Exercise 14.20 Complete the proof of Lemma 14.4.1 (i.e., prove that each *while*$^+$ program can be simulated by a CALC+μ^+ program).

★ **Exercise 14.21** Prove the noninflationary analogue of Lemma 14.4.4 (i.e., that datalog$^{\neg\neg}$ can simulate CALC+μ). *Hint:* Simplify the simulation in Lemma 14.4.4 by taking advantage of the ability to delete in datalog$^{\neg\neg}$. For instance, rules can be inhibited using "switches," which can be turned on and off. Furthermore, no timestamping is needed.

Exercise 14.22 Formulate and prove a normal form for *while*$^+$ and *while*, analogous to the normal forms stated for CALC+μ^+ and CALC+μ.

Exercise 14.23 Prove that RA^+ is equivalent to datalog$^\neg$ and RA is equivalent to noninflationary datalog$^\neg$, and hence to datalog$^{\neg\neg}$. *Hint:* Use Theorems 14.4.5 and 14.4.6 and Exercise 14.16.

Exercise 14.24 Let the *star height* of an RA program be the maximum number of occurrences of * and $^+$ on a path in the syntax tree of the program. Show that each RA program is equivalent to an RA program of star height one.

15 Negation in Datalog

Alice: *I thought we already talked about negation.*
Sergio: *Yes, but they say you don't think by fixpoint.*
Alice: *Humbug, I just got used to it!*
Riccardo: *So we have to tell you how you* really *think.*
Vittorio: *And convince you that our explanation is well founded!*

As originally introduced in Chapter 12, datalog is a toy language that expresses many interesting recursive queries but has serious shortcomings concerning expressive power. Because it is monotonic, it cannot express simple relational algebra queries such as the difference of two relations. In the previous chapter, we considered one approach for adding negation to datalog that led to two procedural languages—namely, inflationary datalog$^\neg$ and datalog$^{\neg\neg}$. In this chapter, we take a different point of view inspired by nonmonotonic reasoning that attempts to view the semantics of such programs in terms of a natural reasoning process.

This chapter begins with illustrations of how the various semantics for datalog do not naturally extend to datalog$^\neg$. Two semantics for datalog$^\neg$ are then considered. The first, called *stratified*, involves a syntactic restriction on programs but provides a semantics that is natural and relatively easy to understand. The second, called *well founded*, requires no syntactic restriction on programs, but the meaning associated with some programs is expressed using a 3-valued logic. (In this logic, facts are true, false, or unknown.) With respect to expressive power, well-founded semantics is equivalent to the *fixpoint* queries, whereas the stratified semantics is strictly weaker. A proof-theoretic semantics for datalog$^\neg$, based on *negation as failure*, is discussed briefly at the end of this chapter.

15.1 The Basic Problem

Suppose that we want to compute the pairs of disconnected nodes in a graph G (i.e., we are interested in the *complement* of the transitive closure of a graph whose edges are given by a binary relation G). We already know how to define the transitive closure of G in a relation T using the datalog program P_{TC} of Chapter 12:

$$T(x, y) \leftarrow G(x, y)$$
$$T(x, y) \leftarrow G(x, z), T(z, y).$$

To define the complement CT of T, we are naturally tempted to use negation as we

did in Chapter 5. Let P_{TCcomp} be the result of adding the following rule to P_{TC}:

$$CT(x, y) \leftarrow \neg T(x, y).$$

To simplify the discussion, we generally assume an active domain interpretation of datalog$^\neg$ rules.

In this example, negation appears to be an appealing addition to the datalog syntax. The language datalog$^\neg$ is defined by allowing, in bodies of rules, literals of the form $\neg R_i(u_i)$, where R_i is a relation name and u_i is a free tuple. In addition, the equality predicate is allowed, and $\neg = (x, y)$ is denoted by $x \neq y$.

One might hope to extend the model-theoretic, fixpoint, and proof-theoretic semantics of datalog just as smoothly as the syntax. Unfortunately, things are less straightforward when negation is present. We illustrate informally the problems that arise if one tries to extend the least-fixpoint and minimal-model semantics of datalog. We shall discuss the proof-theoretic aspect later.

Fixpoint Semantics: Problems

Recall that, for a datalog program P, the fixpoint semantics of P on input $\mathbf{I}$ is the unique minimal fixpoint of the immediate consequence operator T_P containing $\mathbf{I}$. The immediate consequence operator can be naturally extended to a datalog$^\neg$ program P. For a program P, T_P is defined as follows[1]: For each $\mathbf{K}$ over $sch(P)$, A is $T_P(\mathbf{K})$ if $A \in \mathbf{K}|edb(P)$ or if there exists some instantiation $A \leftarrow A_1, \ldots, A_n$ of a rule in P for which (1) if A_i is a positive literal, then $A_i \in \mathbf{K}$; and (2) if $A_i = \neg B_i$ where B_i is a positive literal, then $B_i \notin \mathbf{K}$. [Note the difference from the immediate consequence operator Γ_P defined for datalog$^\neg$ in Section 14.3: Γ_P is inflationary by definition, (that is, $\mathbf{K} \subseteq \Gamma_P(\mathbf{K})$ for each $\mathbf{K}$ over $sch(P)$, whereas T_P is not.] The following example illustrates several unexpected properties that T_P might have.

EXAMPLE 15.1.1

(a) T_P may not have any fixpoint. For the propositional program $P_1 = \{p \leftarrow \neg p\}$, T_{P_1} has no fixpoint.

(b) T_P may have several minimal fixpoints containing a given input. For example, the propositional program $P_2 = \{p \leftarrow \neg q, q \leftarrow \neg p\}$ has two minimal fixpoints (containing the empty instance): $\{p\}$ and $\{q\}$.

(c) Consider the sequence $\{T_P^i(\emptyset)\}_{i>0}$ for a given datalog$^\neg$ program P. Recall that for datalog, the sequence is increasing and converges to the least fixpoint of T_P. In the case of datalog$^\neg$, the situation is more intricate:

1. The sequence does not generally converge, even if T_P has a least fixpoint. For example, let $P_3 = \{p \leftarrow \neg r; r \leftarrow \neg p; p \leftarrow \neg p, r\}$. Then

---

[1] Given an instance $\mathbf{J}$ over a database schema $\mathbf{R}$ with $\mathbf{S} \subseteq \mathbf{R}$, $\mathbf{J}|\mathbf{S}$ denotes the restriction of $\mathbf{J}$ to $\mathbf{S}$.

T_{P_3} has a least fixpoint $\{p\}$ but $\{T^i_{P_3}(\emptyset)\}_{i>0}$ alternates between $\emptyset$ and $\{p, r\}$ and so does not converge (Exercise 15.2).

2. Even if $\{T^i_P(\emptyset)\}_{i>0}$ converges, its limit is not necessarily a minimal fixpoint of T_P, even if such fixpoints exist. To see this, let $P_4 = \{p \leftarrow p, q \leftarrow q, p \leftarrow \neg p, q \leftarrow \neg p\}$. Now $\{T^i_{P_4}(\emptyset)\}_{i>0}$ converges to $\{p, q\}$ but the least fixpoint of T_{P_4} equals $\{p\}$.

Remark 15.1.2 (Inflationary fixpoint semantics) The program P_4 of the preceding example contains two rules of a rather strange form: $p \leftarrow p$ and $q \leftarrow q$. In some sense, such rules may appear meaningless. Indeed, their logical forms [e.g., $(p \vee \neg p)$] are tautologies. However, rules of the form $R(x_1, \ldots, x_n) \leftarrow R(x_1, \ldots, x_n)$ have a nontrivial impact on the immediate consequence operator T_P. If such rules are added for each *idb* relation R, this results in making T_P inflationary [i.e., $\mathbf{K} \subseteq T_P(\mathbf{K})$ for each $\mathbf{K}$], because each fact is an immediate consequence of itself. It is worth noting that in this case, $\{T^i_P(\mathbf{I})\}_{i>0}$ always converges and the semantics given by its limit coincides with the inflationary fixpoint semantics for datalog$^\neg$ programs exhibited in Chapter 14.

To see the difference between the two semantics, consider again program P_{TCcomp}. The sequence $\{T^i_{P_{TCcomp}}(I)\}_{i>0}$ on input I over G converges to the desired answer (the complement of transitive closure). With the inflationary fixpoint semantics, CT becomes a complete graph at the first iteration (because T is initially empty) and P_{TCcomp} does not compute the complement of transitive closure. Nonetheless, it was shown in Chapter 14 that there is a different (more complicated) datalog$^\neg$ program that computes the complement of transitive closure with the inflationary fixpoint semantics. ■

Model-Theoretic Semantics: Problems

As with datalog, we can associate with a datalog$^\neg$ program P the set Σ_P of CALC sentences corresponding to the rules of P. Note first that, as with datalog, Σ_P always has at least one model containing any given input $\mathbf{I}$. $B(P, \mathbf{I})$ is such a model. [Recall that $B(P, \mathbf{I})$, introduced in Chapter 12, is the instance in which the idb relations contain all tuples with values in $\mathbf{I}$ or P.]

For datalog, the model-theoretic semantics of a program P was given by the unique minimal model of Σ_P containing the input. Unfortunately, this simple solution no longer works for datalog$^\neg$, because uniqueness of a minimal model containing the input is not guaranteed. Program P_2 in Example 15.1.1(b) provides one example of this: $\{p\}$ and $\{q\}$ are distinct minimal models of P_2. As another example, consider the program P_{TCcomp} and an input I for predicate G. Let $\mathbf{J}$ over $sch(P_{TCcomp})$ be such that $\mathbf{J}(G) = I$, $\mathbf{J}(T) \supseteq I$, $\mathbf{J}(T)$ is transitively closed, and $\mathbf{J}(CT) = \{\langle x, y\rangle \mid x, y \text{ occur in } \mathbf{I}, \langle x, y\rangle \notin \mathbf{J}(T)\}$. Clearly, there may be more than one such $\mathbf{J}$, but one can verify that each one is a minimal model of $\Sigma_{P_{TCcomp}}$ satisfying $\mathbf{J}(G) = I$.

It is worth noting the connection between T_P and models of Σ_P: An instance $\mathbf{K}$ over $sch(P)$ is a model of Σ_P iff $T_P(\mathbf{K}) \subseteq \mathbf{K}$. In particular, every fixpoint of T_P is a model of Σ_P. The converse is false (Exercise 15.3).

When for a program P, Σ_P has several minimal models, one must specify which

among them is the model intended to be the solution. To this end, various criteria of "niceness" of models have been proposed that can distinguish the intended model from other candidates. We shall discuss several such criteria as we go along. Unfortunately, none of these criteria suffices to do the job. Moreover, upon reflection it is clear that no criteria can exist that would always permit identification of a unique intended model among several minimal models. This is because, as in the case of program P_2 of Example 15.1.1(b), the minimal models can be completely symmetric; in such cases there is no property that would separate one from the others using just the information in the input or the program.

In summary, the approach we used for datalog, based on equivalent least-fixpoint or minimum-model semantics, breaks down when negation is present. We shall describe several solutions to the problem of giving semantics to datalog$^\neg$ programs. We begin with the simplest case and build up from there.

15.2 Stratified Semantics

This section begins with the restricted case in which negation is applied only to *edb* relations. The semantics for negation is straightforward in this case. We then turn to stratified semantics, which extends this simple case in an extremely natural fashion.

Semipositive Datalog$^\neg$

We consider now *semipositive* datalog$^\neg$ programs, which only apply negation to *edb* relations. For example, the difference of R and R' can be defined by the one-rule program

$$Diff(x) \leftarrow R(x), \neg R'(x).$$

To give semantics to $\neg R'(x)$, we simply use the closed world assumption (see Chapter 2): $\neg R'(x)$ holds iff x is in the active domain and $x \notin R'$. Because R' is an *edb* relation, its content is given by the database and the semantics of the program is clear. We elaborate on this next.

Definition 15.2.1 A datalog$^\neg$ program P is *semipositive* if, whenever a negative literal $\neg R'(x)$ occurs in the body of a rule in P, $R' \in edb(P)$.

As their name suggests, semipositive programs are almost positive. One could eliminate negation from semipositive programs by adding, for each *edb* relation R', a new *edb* relation $\overline{R'}$ holding the complement of R' (with respect to the active domain) and replacing $\neg R'(x)$ by $\overline{R'}(x)$. Thus it is not surprising that semipositive programs behave much like datalog programs. The next result is shown easily and is left for the reader (Exercise 15.7).

Theorem 15.2.2 Let P be a semipositive datalog$^\neg$ program. For every instance $\mathbf{I}$ over $edb(P)$,

(i) Σ_P has a unique minimal model $\mathbf{J}$ satisfying $\mathbf{J}|edb(P) = \mathbf{I}$.

(ii) T_P has a unique minimal fixpoint $\mathbf{J}$ satisfying $\mathbf{J}|edb(P) = \mathbf{I}$.

(iii) The minimum model in (i) and the least fixpoint in (ii) are identical and equal to the limit of the sequence $\{T_P^i(\mathbf{I})\}_{i>0}$.

Remark 15.2.3 Observe that in the theorem, we use the formulation "minimal model satisfying $\mathbf{J}|edb(P) = \mathbf{I}$," whereas in the analogous result for datalog we used "minimal model containing $\mathbf{I}$." Both formulations would be equivalent in the datalog setting because adding tuples to the *edb* predicates would result in larger models because of monotonicity. This is not the case here because negation destroys monotonicity. ■

Given a semipositive datalog$^\neg$ program P and an input $\mathbf{I}$, we denote by $P^{semi-pos}(\mathbf{I})$ the minimum model of Σ_P (or equivalently, the least fixpoint of T_P) whose restriction to $edb(P)$ equals $\mathbf{I}$.

An example of a semipositive program that is neither in datalog nor in CALC is given by

$$T(x, y) \leftarrow \neg G(x, y)$$
$$T(x, y) \leftarrow \neg G(x, z), T(z, y).$$

This program computes the transitive closure of the complement of G. On the other hand, the foregoing program for the complement of transitive closure is not a semipositive program. However, it can naturally be viewed as the composition of two semipositive programs: the program computing the transitive closure followed by the program computing its complement. Stratification, which is studied next, may be viewed as the closure of semipositive programs under composition. It will allow us to specify, for instance, the composition just described, computing the complement of transitive closure.

Syntactic Restriction for Stratification

We now consider a natural extension of semipositive programs. In semipositive programs, the use of negation is restricted to *edb* relations. Now suppose that we use some *defined* relations, much like views. Once a relation has been defined by some program, other programs can subsequently treat it as an *edb* relation and apply negation to it. This simple idea underlies an important extension to semipositive programs, called stratified programs.

Suppose we have a datalog$^\neg$ program P. Each *idb* relation is defined by one or more rules of P. If we are able to "read" the program so that, for each *idb* relation R', the portion of P defining R' comes before the negation of R' is used, then we can simply compute R' before its negation is used, and we are done. For example, consider program P_{TCcomp} introduced at the beginning of this chapter. Clearly, we intended for T to be defined by the first two rules before its negation is used in the rule defining CT. Thus the first two rules are applied before the third. Such a way of "reading" P is called a stratification of P and is defined next.

Definition 15.2.4 A *stratification* of a datalog$^\neg$ program P is a sequence of datalog$^\neg$ programs $P^1, \ldots, P^n$ such that for some mapping σ from $idb(P)$ to $[1..n]$,

(i) $\{P^1, \ldots, P^n\}$ is a partition of P.

(ii) For each predicate R, all the rules in P defining R are in $P^{\sigma(R)}$ (i.e., in the same program of the partition).

(iii) If $R(u) \leftarrow \ldots R'(v) \ldots$ is a rule in P, and R' is an *idb* relation, then $\sigma(R') \leq \sigma(R)$.

(iv) If $R(u) \leftarrow \ldots \neg R'(v) \ldots$ is a rule in P, and R' is an *idb* relation, then $\sigma(R') < \sigma(R)$.

Given a stratification $P^1, \ldots, P^n$ of P, each P^i is called a *stratum* of the stratification, and σ is called the *stratification mapping*.

Intuitively, a stratification of a program P provides a way of parsing P as a sequence of subprograms $P^1, \ldots, P^n$, each defining one or several *idb* relations. By (iii), if a relation R' is used positively in the definition of R, then R' must be defined earlier or simultaneously with R (this allows recursion!). If the negation of R' is used in the definition of R, then by (iv) the definition of R' must come strictly before that of R.

Unfortunately, not every datalog$^\neg$ program has a stratification. For example, there is no way to "read" program P_2 of Example 15.1.1 so that p is defined before q and q before p. Programs that have a stratification are called *stratifiable*. Thus P_2 is not stratifiable. On the other hand, P_{TCcomp} is clearly stratifiable: The first stratum consists of the first two rules (defining T), and the second stratum consists of the third rule (defining CT using T).

Example 15.2.5 Consider the program P_7 defined by

$$\begin{aligned} r_1 \quad & S(x) \leftarrow R'_1(x), \neg R(x) \\ r_2 \quad & T(x) \leftarrow R'_2(x), \neg R(x) \\ r_3 \quad & U(x) \leftarrow R'_3(x), \neg T(x) \\ r_4 \quad & V(x) \leftarrow R'_4(x), \neg S(x), \neg U(x). \end{aligned}$$

Then P_7 has 5 distinct stratifications, namely,

$$\begin{aligned} &\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\} \\ &\{r_2\}, \{r_1\}, \{r_3\}, \{r_4\} \\ &\{r_2\}, \{r_3\}, \{r_1\}, \{r_4\} \\ &\{r_1, r_2\}, \{r_3\}, \{r_4\} \\ &\{r_2\}, \{r_1, r_3\}, \{r_4\}. \end{aligned}$$

These lead to five different ways of reading the program P_7. As will be seen, each of these yields the same semantics.

There is a simple test for checking if a program is stratifiable. Not surprisingly, it involves testing for an acyclicity condition in definitions of relations using negation. Let P be a datalog$^\neg$ program. The *precedence graph* G_P of P is the labeled graph whose nodes are the *idb* relations of P. Its edges are the following:

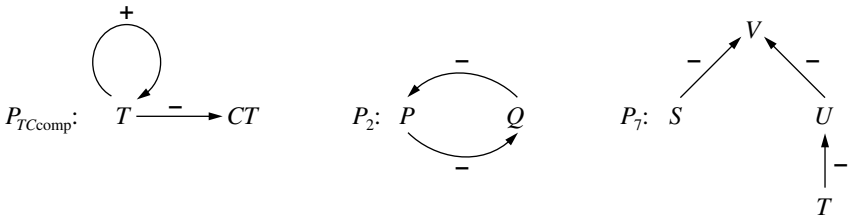


Figure 15.1: Precedence graphs for P_{CT}, P_2, and P_7

- If $R(u) \leftarrow \ldots R'(v) \ldots$ is a rule in P, then $\langle R', R\rangle$ is an edge in G_P with label $+$ (called a *positive edge*).
- If $R(u) \leftarrow \ldots \neg R'(v) \ldots$ is a rule in P, then $\langle R', R\rangle$ is an edge in G_P with label $-$ (called a *negative edge*).

For example, the precedence graphs for program $P_{TC\,comp}$, P_2, and P_7 are represented in Fig. 15.1. It is straightforward to show the following (proof omitted):

Lemma 15.2.6 Let P be a program with stratification σ. If there is a path from R' to R in G_P, then $\sigma(R') \leq \sigma(R)$; and if there is a path from R' to R in G_P containing some negative edge, then $\sigma(R') < \sigma(R)$.

We now show how the precedence graph of a program can be used to test the stratifiability of the program.

Proposition 15.2.7 A datalog$^\neg$ program P is stratifiable iff its precedence graph G_P has no cycle containing a negative edge.

Proof Consider the "only if" part. Suppose P is a datalog$^\neg$ program whose precedence graph has a cycle $R_1, \ldots R_m, R_1$ containing a negative edge, say from R_m to R_1. Suppose, toward a contradiction, that σ is a stratification mapping for P. By Lemma 15.2.6, $\sigma(R_1) < \sigma(R_1)$, because there is a path from R_1 to R_1 with a negative edge. This is a contradiction, so no stratification mapping σ exists for P.

Conversely, suppose P is a program whose precedence graph G_P has no cycle with negative edges. Let $\prec$ be the binary relation among the strongly connected components of G_P defined as follows: $C \prec C'$ if $C \neq C'$ and there is a (positive or negative) edge in G_P from some node of C to some node of C'.

We first show that

(*) $\prec$ is acyclic.

Suppose there is a cycle in $\prec$. Then by construction of $\prec$, this cycle must traverse two *distinct* strongly connected components, say C, C'. Let A be in C. It is easy to deduce that there is a path in G_P from some vertex in C' to A and from A to some vertex in C'.

Because C' is a strongly connected component of G_P, A is in C'. Thus $C \subseteq C'$, so $C = C'$, a contradiction. Hence (*) holds.

In view of (*), the binary relation $\prec$ induces a partial order among the strongly connected components of G_P, which we also denote by $\prec$, by abuse of notation. Let $C^1, \ldots, C^n$ be a topographic sort with respect to $\prec$ of the strongly connected components of G_P; that is, $C^1 \ldots C^n$ is the set of strongly connected components of G_P and if $C^i \prec C^j$, then $i \leq j$. Finally, for each i, $1 \leq i \leq n$, let Q^i consist of all rules defining some relation in C^i. Then $Q^1, \ldots, Q^n$ is a stratification of P. Indeed, (i) and (ii) in the definition of stratification are clearly satisfied. Conditions (iii) and (iv) follow immediately from the construction of G_P and $\prec$ and from the hypothesis that G_P has no cycle with negative edge. ■

Clearly, the stratifiability test provided by Proposition 15.2.7 takes time polynomial in the size of the program P.

Verification of the following observation is left to the reader (Exercise 15.4).

Lemma 15.2.8 Let $P^1, \ldots, P^n$ be a stratification of P, and let $Q^1, \ldots, Q^m$ be obtained as in Proposition 15.2.7. If $Q^j \cap P^i \neq \emptyset$, then $Q^j \subseteq P^i$. In particular, the partition $Q^1, \ldots, Q^m$ of P refines all other partitions given by stratifications of P.

Semantics of Stratified Programs

Consider a stratifiable program P with a stratification $\sigma = P^1, \ldots, P^n$. Using the stratification σ, we can now easily give a semantics to P using the well-understood semipositive programs. Notice that for each program P^i in the stratification, if P^i uses the negation of R', then $R' \in edb(P^i)$ [note that $edb(P^i)$ may contain some of the *idb* relations of P]. Furthermore, R' is either in $edb(P)$ or is defined by some P^j preceding P^i [i.e., $R' \in \cup_{j<i} idb(P^j)$]. Thus each program P^i is semipositive relative to previously defined relations. Then the semantics of P is obtained by applying, in order, the programs P^i. More precisely, let $\mathbf{I}$ be an instance over $edb(P)$. Define the sequence of instances

$$\mathbf{I}_0 = \mathbf{I}$$
$$\mathbf{I}_i = \mathbf{I}_{i-1} \cup P^i(\mathbf{I}_{i-1}|edb(P^i)), 0 < i \leq n.$$

Note that $\mathbf{I}_i$ extends $\mathbf{I}_{i-1}$ by providing values to the relations defined by P^i; and that $P^i(\mathbf{I}_{i-1}|edb(P^i))$, or equivalently, $P^i(\mathbf{I}_{i-1})$, is the semantics of the semipositive program P^i applied to the values of its *edb* relations provided by $\mathbf{I}_{i-1}$. Let us denote the final instance $\mathbf{I}_n$ thus obtained by $\sigma(\mathbf{I})$. This provides the *semantics of* a datalog$^\neg$ program *under a stratification* σ.

Independence of Stratification

As shown in Example 15.2.5, a datalog$^\neg$ program can have more than one stratification. Will the different stratifications yield the same semantics? Fortunately, the answer is yes.

To demonstrate this, we use the following simple lemma, whose proof is left to the reader (Exercise 15.10).

LEMMA 15.2.9 Let P be a semipositive datalog$^\neg$ program and σ a stratification for P. Then $P^{semi-pos}(\mathbf{I}) = \sigma(\mathbf{I})$ for each instance $\mathbf{I}$ over $edb(P)$.

Two stratifications of a datalog$^\neg$ program are *equivalent* if they yield the same semantics on all inputs.

THEOREM 15.2.10 Let P be a stratifiable datalog$^\neg$ program. All stratifications of P are equivalent.

Proof Let G_P be the precedence graph of P and $\sigma_{G_P} = Q^1, \ldots, Q^n$ be a stratification constructed from G_P as in the proof of Theorem 15.2.7. Let $\sigma = P^1, \ldots, P^k$ be a stratification of P. It clearly suffices to show that σ is equivalent to σ_{G_P}. The stratification σ_{G_P} is used as a reference because, as shown in Lemma 15.2.8, its strata are the finest possible among all stratifications for P.

As in the proof of Theorem 15.2.7, we use the partial order $\prec$ among the strongly connected components of G_P and the notation introduced there. Clearly, the relation $\prec$ on the C^i induces a partial order on the Q^i, which we also denote by $\prec$ ($Q^i \prec Q^j$ if $C^i \prec C^j$). We say that a sequence $Q^{i_1}, \ldots, Q^{i_r}$ of some of the Q^i is *compatible* with $\prec$ if for every $l < m$ it is *not* the case that $Q^{i_m} \prec Q^{i_l}$.

We shall prove that

1. If σ' and σ'' are permutations of σ_{G_P} that are compatible with $\prec$, then σ' and σ'' are equivalent stratifications of P.
2. For each P^i, $1 \leq i \leq k$, there exists $\sigma_i = Q^{i_1}, \ldots, Q^{i_r}$ such that σ_i is a stratification of P^i, and the sequence $Q^{i_1}, \ldots, Q^{i_r}$ is compatible with $\prec$.
3. $\sigma_1, \ldots, \sigma_k$ is a permutation of $Q^1, \ldots, Q^n$ compatible with $\prec$.

Before demonstrating these, we argue that the foregoing statements (1 through 3) are sufficient to show that σ and σ_{G_P} are equivalent. By statement 2, each σ_i is a stratification of P^i. Lemma 15.2.9 implies that P^i is equivalent to σ_i. It follows that $\sigma = P^1, \ldots, P^k$ is equivalent to $\sigma_1, \ldots, \sigma_k$ which, by statement 3, is a permutation of σ_{G_P} compatible with $\prec$. Then $\sigma_1, \ldots, \sigma_k$ and σ_{G_P} are equivalent by statement 1, so σ and σ_{G_P} are equivalent.

Consider statement 1. Note first that one can obtain σ'' from σ' by a sequence of exchanges of adjacent Q^i, Q^j such that $Q^i \not\prec Q^j$ and $Q^j \not\prec Q^i$ (Exercise 15.9). Thus it is sufficient to show that for every such pair, Q^i, Q^j is equivalent to Q^j, Q^i. Because $Q^i \not\prec Q^j$ and $Q^j \not\prec Q^i$, it follows that no *idb* relation of Q^i occurs in Q^j and conversely. Then $Q^i \cup Q^j$ is a semipositive program [with respect to $edb(Q^i \cup Q^j)$] and both Q^i, Q^j and Q^j, Q^i are stratifications of $Q^i \cup Q^j$. By Lemma 15.2.9, Q^i, Q^j and Q^j, Q^i are both equivalent to $Q^i \cup Q^j$ (as a semipositive program), so Q^i, Q^j and Q^j, Q^i are equivalent.

Statement 2 follows immediately from Lemma 15.2.8.

Finally, consider statement 3. By statement 2, each σ_i is compatible with $\prec$. Thus it remains to be shown that, if Q^m occurs in σ_i, Q^l occurs in σ_j, and $i < j$, then $Q^l \not\prec Q^m$.

Note that Q^l is included in P^j, and Q^m is included in P^i. It follows that for all relations R defined by Q^m and R' defined by Q^l, $\sigma(R) < \sigma(R')$, where σ is the stratification function of $P^1, \ldots, P^k$. Hence $R' \not\prec R$ so $Q^l \not\prec Q^m$. ■

Thus all stratifications of a given stratifiable program are equivalent. This means that we can speak about the semantics of such a program independently of a particular stratification. Given a stratifiable datalog$^\neg$ program P and an input $\mathbf{I}$ over $edb(P)$, we shall take as the semantics of P on $\mathbf{I}$ the semantics $\sigma(\mathbf{I})$ of any stratification σ of P. This semantics, well defined by Theorem 15.2.10, is denoted by $P^{strat}(\mathbf{I})$. Clearly, $P^{strat}(\mathbf{I})$ can be computed in time polynomial with respect to $\mathbf{I}$.

Now that we have a well-defined semantics for stratified programs, we can verify that for semipositive programs, the semantics coincides with the semantics already introduced. If P is a semipositive datalog$^\neg$ program, then P is also stratifiable. By Lemma 15.2.9, $P^{semi-pos}$ and P^{strat} are equivalent.

Properties of Stratified Semantics

Stratified semantics has a procedural flavor because it is the result of an ordering of the rules, albeit implicit. What can we say about $P^{strat}(\mathbf{I})$ from a model-theoretic point of view? Rather pleasantly, $P^{strat}(\mathbf{I})$ is a minimal model of Σ_P containing $\mathbf{I}$. However, no precise characterization of stratified semantics in model-theoretic terms has emerged. Some model-theoretic properties of stratified semantics are established next.

Proposition 15.2.11 For each stratifiable datalog$^\neg$ program P and instance $\mathbf{I}$ over $edb(\mathbf{I})$,

(a) $P^{strat}(\mathbf{I})$ is a minimal model of Σ_P whose restriction to $edb(P)$ equals $\mathbf{I}$.

(b) $P^{strat}(\mathbf{I})$ is a minimal fixpoint of T_P whose restriction to $edb(P)$ equals $\mathbf{I}$.

Proof For part (a), let $\sigma = P^1, \ldots, P^n$ be a stratification of P and $\mathbf{I}$ an instance over $edb(P)$. We have to show that $P^{strat}(\mathbf{I})$ is a minimal model of Σ_P whose restriction to $edb(P)$ equals $\mathbf{I}$. Clearly, $P^{strat}(\mathbf{I})$ is a model of Σ_P whose restriction to $edb(P)$ equals $\mathbf{I}$. To prove its minimality, it is sufficient to show that, for each model $\mathbf{J}$ of Σ_P,

$$(**) \qquad \text{if } \mathbf{I} \subseteq \mathbf{J} \subseteq P^{strat}(\mathbf{I}) \text{ then } \mathbf{J} = P^{strat}(\mathbf{I}).$$

Thus suppose $\mathbf{I} \subseteq \mathbf{J} \subseteq P^{strat}(\mathbf{I})$. We prove by induction on k that

$$(\dagger) \qquad P^{strat}(\mathbf{I})|sch(\cup_{i \le k} P^i) = \mathbf{J}|sch(\cup_{i \le k} P^i)$$

for each k, $1 \le k \le n$. The equality of $P^{strat}(\mathbf{I})$ and $\mathbf{J}$ then follows from ($\dagger$) with $k = n$.

For $k = 1$, $edb(P^1) \subseteq edb(P)$ so

$$P^{strat}(\mathbf{I})|edb(P^1) = \mathbf{I}|edb(P^1) = \mathbf{J}|edb(P^1).$$

By the definition of stratified semantics and Theorem 15.2.2, $P^{strat}(\mathbf{I})|sch(P^1)$ is the

minimum model of Σ_{P^1} whose restriction to $edb(P^1)$ equals $P^{strat}(\mathbf{I})|edb(P^1)$. On the other hand, $\mathbf{J}|sch(P^1)$ is also a model of Σ_{P^1} whose restriction to $edb(P^1)$ equals $P^{strat}(\mathbf{I})|edb(P^1)$. From the minimality of $P^{strat}(\mathbf{I})|sch(P^1)$, it follows that

$$P^{strat}(\mathbf{I})|sch(P^1) \subseteq \mathbf{J}|sch(P^1).$$

From (**) it then follows that $P^{strat}(\mathbf{I})|sch(P^1) = \mathbf{J}|sch(P^1)$, which establishes (†) for $k = 1$. For the induction step, suppose (†) is true for k, $1 \leq k < n$. Then (†) for $k + 1$ is shown in the same manner as for the case $k = 1$. This proves (†) for $1 \leq k \leq n$. It follows that $P^{strat}(\mathbf{I})$ is a minimal model of Σ_P whose restriction to $edb(P)$ equals $\mathbf{I}$.

The proof of part (b) is left for Exercise 15.12. ■

There is another appealing property of stratified semantics that takes into account the syntax of the program in addition to purely model-theoretic considerations. This property is illustrated next.

Consider the two programs

$$P_5 = \{p \leftarrow \neg q\}$$
$$P_6 = \{q \leftarrow \neg p\}$$

From the perspective of classical logic, Σ_{P_5} and Σ_{P_6} are equivalent to each other and to $\{p \vee q\}$. However, T_{P_5} and T_{P_6} have different behavior: The unique fixpoint of T_{P_5} is $\{p\}$, whereas that of T_{P_6} is $\{q\}$. This is partially captured by the notion of "supported" as follows.

Let datalog$^\neg$ program P and input $\mathbf{I}$ be given. As with pure datalog, $\mathbf{J}$ is a model of P iff $\mathbf{J} \supseteq T_P(\mathbf{J})$. We say that $\mathbf{J}$ is a *supported* model if $\mathbf{J} \subseteq T_P(\mathbf{J})$ (i.e., if each fact in $\mathbf{J}$ is "justified" or supported by being the head of a ground instantiation of a rule in P whose body is all true in $\mathbf{J}$). (In the context of some input $\mathbf{I}$, we say that $\mathbf{J}$ is supported *relative* to $\mathbf{I}$ and the definition is modified accordingly.) This condition, which has both syntactic and semantic aspects, captures at least some of the spirit of the immediate consequence operator T_P. As suggested in Remark 15.1.2, its impact can be annulled by adding rules of the form $p \leftarrow p$.

The proof of the following is left to the reader (Exercise 15.13).

Proposition 15.2.12 For each stratifiable program P and instance $\mathbf{I}$ over $edb(P)$, $P^{strat}(\mathbf{I})$ is a supported model of P relative to $\mathbf{I}$.

We have seen that stratification provides an elegant and simple approach to defining semantics of datalog$^\neg$ programs. Nonetheless, it has two major limitations. First, it does not provide semantics to *all* datalog$^\neg$ programs. Second, stratified datalog$^\neg$ programs are not entirely satisfactory with regard to expressive power. From a computational point of view, they provide recursion and negation and are inflationary. Therefore, as discussed in Chapter 14, one might expect that they express the *fixpoint* queries. Unfortunately, stratified datalog$^\neg$ programs fall short of expressing all such queries, as will be shown in Section 15.4. Intuitively, this is because the stratification condition prohibits recursive

application of negation, whereas in other languages expressing *fixpoint* this computational restriction does not exist.

For these reasons, we consider another semantics for datalog$^\neg$ programs called *well founded*. As we shall see, this provides semantics to all datalog$^\neg$ programs and expresses all *fixpoint* queries. Furthermore, well-founded and stratified semantics agree on stratified datalog$^\neg$ programs.

15.3 Well-Founded Semantics

Well-founded semantics relies on a fundamental revision of our expectations of the answer to a datalog$^\neg$ program. So far, we required that the answer must provide information on the truth or falsehood of every fact. Well-founded semantics is based on the idea that a given program may not necessarily provide such information on all facts. Instead some facts may simply be indifferent to it, and the answer should be allowed to say that the truth value of those facts is *unknown*. As it turns out, relaxing expectations about the answer in this fashion allows us to provide a natural semantics for *all* datalog$^\neg$ programs. The price is that the answer is no longer guaranteed to provide total information.

Another aspect of this approach is that it puts negative and positive facts on a more equal footing. One can no longer assume that $\neg R(u)$ is true simply because $R(u)$ is not in the answer. Instead, both negative and positive facts must be inferred. To formalize this, we shall introduce 3-valued instances, in which the truth value of facts can be **true**, **false**, or **unknown**.

This section begins by introducing a largely declarative semantics for datalog$^\neg$ programs. Next an equivalent fixpoint semantics is developed. Finally it is shown that stratified and well-founded semantics agree on the family of stratified datalog$^\neg$ programs.

A Declarative Semantics for Datalog$^\neg$

The aim of giving semantics to a datalog$^\neg$ program P will be to find an appropriate 3-valued model $\mathbf{I}$ of Σ_P. In considering what *appropriate* might mean, it is useful to recall the basic motivation underlying the logic-programming approach to negation as opposed to the purely computational approach. An important goal is to model some form of natural reasoning process. In particular, consistency in the reasoning process is required. Specifically, one cannot use a fact and later infer its negation. This should be captured in the notion of appropriateness of a 3-valued model $\mathbf{I}$, and it has two intuitive aspects:

- the positive facts of $\mathbf{I}$ must be inferred from P assuming the negative facts in $\mathbf{I}$; and
- all negative facts that can be inferred from $\mathbf{I}$ must already be in $\mathbf{I}$.

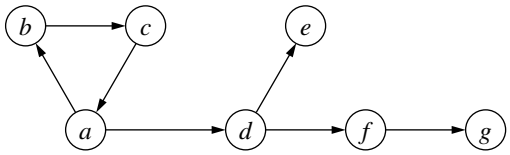
A 3-valued model satisfying the aforementioned notion of appropriateness will be called a 3-stable model of P. It turns out that, generally, programs have several 3-stable models. Then it is natural to take as an answer the certain (positive and negative) facts that belong to all such models, which turns out to yield, in some sense, the smallest 3-stable model. This is indeed how the well-founded semantics of P will be defined.

EXAMPLE 15.3.1 The example concerns a game with states, $a, b, \ldots$. The game is between two players. The possible moves of the games are held in a binary relation *moves*. A tuple $\langle a, b\rangle$ in *moves* indicates that when in state a, one can choose to move to state b. A player loses if he or she is in a state from which there are no moves. The goal is to compute the set of winning states (i.e., the set of states such that there exists a winning strategy for a player in this state). These are obtained in a unary predicate *win*.

Consider the input **K** with the following value for *moves*:

$$\mathbf{K}(moves) = \{\langle b, c\rangle, \langle c, a\rangle, \langle a, b\rangle, \langle a, d\rangle, \langle d, e\rangle, \langle d, f\rangle, \langle f, g\rangle\}$$

Graphically, the input is represented as



It is seen easily that there are indeed winning strategies from states d (move to e) and f (move to g). Slightly more subtle is the fact that there is no winning strategy from any of states a, b, or c. A given player can prevent the other from winning, essentially by forcing a nonterminating sequence of moves.

Now consider the following nonstratifiable program P_{win}:

$$win(x) \leftarrow moves(x, y), \neg win(y)$$

Intuitively, P_{win} states that a state x is in *win* if there is at least one state y that one can move to from x, for which the opposing player loses. We now exhibit a 3-valued model **J** of P_{win} that agrees with **K** on *moves*. As will be seen, this will in fact be the well-founded semantics of P_{win} on input **K**. Instance **J** is such that $\mathbf{J}(moves) = \mathbf{K}(moves)$ and the values of *win*-atoms are given as follows:

true	$win(d), win(f)$
false	$win(e), win(g)$
unknown	$win(a), win(b), win(c)$

We now embark on defining formally the well-founded semantics. We do this in three steps. First we define the notion of 3-valued instance and extend the notion of truth value and satisfaction. Then we consider datalog and show the existence of a minimum 3-valued model for each datalog program. Finally we consider datalog$^{\neg}$ and the notion of 3-stable model, which is the basis of well-founded semantics.

3-valued Instances Dealing with three truth values instead of the usual two requires extending some of the basic notions like instance and model. As we shall see, this is straightforward. We will denote **true** by 1, **false** by 0, and **unknown** by 1/2.

Consider a datalog$^\neg$ program P and a classical 2-valued instance $\mathbf{I}$. As was done in the discussion of SLD resolution in Chapter 12, we shall denote by $P_\mathbf{I}$ the program obtained from P by adding to P unit clauses stating that the facts in $\mathbf{I}$ are true. Then $P(\mathbf{I}) = P_\mathbf{I}(\emptyset)$. For the moment, we shall deal with datalog$^\neg$ programs such as these, whose input is included in the program. Recall that $\mathbf{B}(P)$ denotes all facts of the form $R(a_1, \ldots, a_k)$, where R is a relation and $a_1, \ldots, a_k$ constants occurring in P. In particular, $\mathbf{B}(P_\mathbf{I}) = \mathbf{B}(P, \mathbf{I})$.

Let P be a datalog$^\neg$ program. A *3-valued instance* $\mathbf{I}$ over $sch(P)$ is a total mapping from $\mathbf{B}(P)$ to $\{0, 1/2, 1\}$. We denote by $\mathbf{I}^1$, $\mathbf{I}^{1/2}$, and $\mathbf{I}^0$ the set of atoms in $\mathbf{B}(P)$ whose truth value is 1, $1/2$, and 0, respectively. A 3-valued instance $\mathbf{I}$ is *total*, or *2-valued*, if $\mathbf{I}^{1/2} = \emptyset$. There is a natural ordering $\prec$ among 3-valued instances over $sch(P)$, defined by

$$\mathbf{I} \prec \mathbf{J} \text{ iff for each } A \in \mathbf{B}(P),\ \ \mathbf{I}(A) \leq \mathbf{J}(A).$$

Note that this is equivalent to $\mathbf{I}^1 \subseteq \mathbf{J}^1$ and $\mathbf{I}^0 \supseteq \mathbf{J}^0$ and that it generalizes containment for 2-valued instances.

Occasionally, we will represent a 3-valued instance by listing the positive and negative facts and omitting the undefined ones. For example, the 3-valued instance $\mathbf{I}$, where $\mathbf{I}(p) = 1, \mathbf{I}(q) = 1, \mathbf{I}(r) = 1/2, \mathbf{I}(s) = 0$, will also be written as $\mathbf{I} = \{p, q, \neg s\}$.

Given a 3-valued instance $\mathbf{I}$, we next define the truth value of Boolean combinations of facts using the connectives $\vee, \wedge, \neg, \leftarrow$. The truth value of a Boolean combination α of facts is denoted by $\hat{\mathbf{I}}(\alpha)$, defined by

$$\begin{aligned}
\hat{\mathbf{I}}(\beta \wedge \gamma) &= \min\{\hat{\mathbf{I}}(\beta), \hat{\mathbf{I}}(\gamma)\} \\
\hat{\mathbf{I}}(\beta \vee \gamma) &= \max\{\hat{\mathbf{I}}(\beta), \hat{\mathbf{I}}(\gamma)\} \\
\hat{\mathbf{I}}(\neg\beta) &= 1 - \hat{\mathbf{I}}(\beta) \\
\hat{\mathbf{I}}(\beta \leftarrow \gamma) &= 1 \text{ if } \hat{\mathbf{I}}(\gamma) \leq \hat{\mathbf{I}}(\beta), \text{ and 0 otherwise.}
\end{aligned}$$

The reader should be careful: Known facts about Boolean operators in the 2-valued context may not hold in this more complex one. For instance, note that the truth value of $p \leftarrow q$ may be different from that of $p \vee \neg q$ (see Exercise 15.15). To see that the preceding definition matches the intuition, one might want to verify that with the specific semantics of $\leftarrow$ used here, the instance $\mathbf{J}$ of Example 15.3.1 does satisfy (the ground instantiation of) $P_{win,\mathbf{K}}$. That would not be the case if we define the semantics of $\leftarrow$ in a more standard way; by using $p \leftarrow q \equiv p \vee \neg q$.

A 3-valued instance $\mathbf{I}$ over $sch(P)$ *satisfies* a Boolean combination α of atoms in $\mathbf{B}(P)$ iff $\hat{\mathbf{I}}(\alpha) = 1$. Given a datalog$^{(\neg)}$ program P, a *3-valued model* of Σ_P is a 3-valued instance over $sch(P)$ satisfying the set of implications corresponding to the rules in $ground(P)$.

Example 15.3.2 Recall the program P_{win} of Example 15.3.1 and the input instance $\mathbf{K}$ and output instance $\mathbf{J}$ presented there. Consider these ground sentences:

$$win(a) \leftarrow moves(a, d), \neg win(d)$$
$$win(a) \leftarrow moves(a, b), \neg win(b).$$

The first is true for $\mathbf{J}$, because $\hat{\mathbf{J}}(\neg win(d)) = 0$, $\hat{\mathbf{J}}(moves(a, d)) = 1$, $\hat{\mathbf{J}}(win(a)) = 1/2$, and $1/2 \geq 0$. The second is true because $\hat{\mathbf{J}}(\neg win(b)) = 1/2$, $\hat{\mathbf{J}}(moves(a, b)) = 1$, $\hat{\mathbf{J}}(win(a)) = 1/2$, and $1/2 \geq 1/2$.

Observe that, on the other hand,

$$\hat{\mathbf{J}}(win(a) \vee \neg(moves(a, b) \wedge \neg win(b))) = 1/2.$$

3-valued Minimal Model for Datalog We next extend the definition and semantics of datalog programs to the context of 3-valued instances. Although datalog programs do not contain negation, they will now be allowed to infer positive, unknown, and false facts. The syntax of a *3-extended datalog program* is the same as for datalog, except that the truth values $0, 1/2$, and 1 can occur as literals in bodies of rules. Given a 3-extended datalog program P, the *3-valued immediate consequence operator* $3\text{-}T_P$ of P is a mapping on 3-valued instances over $sch(P)$ defined as follows. Given a 3-valued instance $\mathbf{I}$ and $A \in \mathbf{B}(P)$, $3\text{-}T_P(\mathbf{I})(A)$ is

1 if there is a rule $A \leftarrow body$ in $ground(P)$ such that $\hat{\mathbf{I}}(body) = 1$,

0 if for each rule $A \leftarrow body$ in $ground(P)$, $\hat{\mathbf{I}}(body) = 0$ (and, in particular, if there is no rule with A in head),

$1/2$ otherwise.

Example 15.3.3 Consider the 3-extended datalog program $P = \{p \leftarrow 1/2;\ p \leftarrow q, 1/2;\ q \leftarrow p, r;\ q \leftarrow p, s;\ s \leftarrow q;\ r \leftarrow 1\}$. Then

$$\begin{aligned} 3\text{-}T_P(\{\neg p, \neg q, \neg r, \neg s\}) &= \{\neg q, r, \neg s\} \\ 3\text{-}T_P(\{\neg q, r, \neg s\}) &= \{r, \neg s\} \\ 3\text{-}T_P(\{r, \neg s\}) &= \{r\} \\ 3\text{-}T_P(\{r\}) &= \{r\}. \end{aligned}$$

In the following, 3-valued instances are compared with respect to $\prec$. Thus "least," "minimal," and "monotonic" are with respect to $\prec$ rather than the set inclusion used for classical 2-valued instances. In particular, note that the minimum 3-valued instance with respect to $\prec$ is that where all atoms are false. Let $\perp$ denote this particular instance.

With the preceding definitions, extended datalog programs on 3-valued instances behave similarly to classical programs. The next lemma can be verified easily (Exercise 15.16):

Lemma 15.3.4 Let P be a 3-extended datalog program. Then

1. $3{-}T_P$ is monotonic and the sequence $\{3\text{-}T_P^i(\perp)\}_{i>0}$ is increasing and converges to the least fixpoint of $3\text{-}T_P$;

2. P has a unique minimal 3-valued model that equals the least fixpoint of $3\text{-}T_P$.

The semantics of an extended datalog program is the minimum 3-valued model of P. Analogous to conventional datalog, we denote this by $P(\perp)$.

3-stable Models of Datalog$^\neg$

We are now ready to look at datalog$^\neg$ programs and formally define 3-stable models of a datalog$^\neg$ program P. We "bootstrap" to the semantics of programs with negation, using the semantics for 3-extended datalog programs described earlier. Let $\mathbf{I}$ be a 3-valued instance over $sch(P)$. We reduce the problem to that of applying a positive datalog program, as follows. The *positivized ground version* of P given $\mathbf{I}$, denoted $pg(P, \mathbf{I})$, is the 3-extended datalog program obtained from $ground(P)$ by replacing each negative premise $\neg A$ by $\hat{\mathbf{I}}(\neg A)$ (i.e., 0, 1, or 1/2). Because all negative literals in $ground(P)$ have been replaced by their truth value in $\mathbf{I}$, $pg(P, \mathbf{I})$ is now a 3-extended datalog program (i.e, a program without negation). Its least fixpoint $pg(P, \mathbf{I})(\perp)$ contains all the facts that are consequences of P by assuming the values for the negative premises as given by $\mathbf{I}$. We denote $pg(P, \mathbf{I})(\perp)$ by $conseq_P(\mathbf{I})$. Thus the intuitive conditions required of 3-stable models now amount to $conseq_P(\mathbf{I}) = \mathbf{I}$.

DEFINITION 15.3.5 Let P be a datalog$^\neg$ program. A 3-valued instance $\mathbf{I}$ over $sch(P)$ is a *3-stable model* of P iff $conseq_P(\mathbf{I}) = \mathbf{I}$.

Observe an important distinction between $conseq_P$ and the immediate consequence operator used for inflationary datalog$^\neg$. For inflationary datalog$^\neg$, we assumed that $\neg A$ was true as long as A was not inferred. Here we just assume in such a case that A is unknown and try to prove new facts. Of course, doing so requires the 3-valued approach.

EXAMPLE 15.3.6 Consider the following datalog$^\neg$ program P:

$$
\begin{aligned}
p &\leftarrow \neg r \\
q &\leftarrow \neg r,\ p \\
s &\leftarrow \neg t \\
t &\leftarrow q,\ \neg s \\
u &\leftarrow \neg t,\ p,\ s
\end{aligned}
$$

The program has three 3-stable models (represented by listing the positive and negative facts and leaving out the unknown facts):

$$
\begin{aligned}
\mathbf{I}_1 &= \{p, q, t, \neg r, \neg s, \neg u\} \\
\mathbf{I}_2 &= \{p, q, s, \neg r, \neg t, \neg u\} \\
\mathbf{I}_3 &= \{p, q, \neg r\}
\end{aligned}
$$

Let us check that $\mathbf{I}_3$ is a 3-stable model of P. The program $P' = pg(P, \mathbf{I}_3)$ is

$$p \leftarrow 1$$
$$q \leftarrow 1, p$$
$$s \leftarrow 1/2$$
$$t \leftarrow q, 1/2$$
$$u \leftarrow 1/2, p, s$$

The minimum 3-valued model of $pg(P, \mathbf{I}_3)$ is obtained by iterating $3\text{-}T_{P'}(\bot)$ up to a fixpoint. Thus we start with $\bot = \{\neg p, \neg q, \neg r, \neg s, \neg t, \neg u\}$. The first application of $3\text{-}T_{P'}$ yields $3\text{-}T_{P'}(\bot) = \{p, \neg q, \neg r, \neg t, \neg u\}$. Next $(3\text{-}T_{P'})^2(\bot) = \{p, q, \neg r, \neg t\}$. Finally $(3\text{-}T_{P'})^3(\bot) = (3\text{-}T_{P'})^4(\bot) = \{p, q, \neg r\}$. Thus

$$conseq_P(\mathbf{I}_3) = pg(P, \mathbf{I}_3)(\bot) = (3\text{-}T_{P'})^3(\bot) = \mathbf{I}_3,$$

and $\mathbf{I}_3$ is a 3-stable model of P.

The reader is invited to verify that in Example 15.3.1, the instance $\mathbf{J}$ is a 3-stable model of the program $P_{win,\mathbf{K}}$ for the input instance $\mathbf{K}$ presented there.

As seen from the example, datalog$^\neg$ programs generally have several 3-stable models. We will show later that each datalog$^\neg$ program has at least one 3-stable model. Therefore it makes sense to let the final answer consist of the positive and negative facts belonging to all 3-stable models of the program. As we shall see, the 3-valued instance so obtained is itself a 3-stable model of the program.

DEFINITION 15.3.7 Let P be a datalog$^\neg$ program. The *well-founded semantics* of P is the 3-valued instance consisting of all positive and negative facts belonging to all 3-stable models of P. This is denoted by $P^{wf}(\emptyset)$,or simply, P^{wf}. Given datalog$^\neg$ program P and input instance $\mathbf{I}$, $P^{wf}_{\mathbf{I}}(\emptyset)$ is denoted $P^{wf}(\mathbf{I})$.

Thus the well-founded semantics of the program P in Example 15.3.6 is $P^{wf}(\emptyset) = \{p, q, \neg r\}$. We shall see later that in Example 15.3.1, $P^{wf}_{win}(\mathbf{K}) = \mathbf{J}$.

A Fixpoint Definition

Note that the preceding description of the well-founded semantics, although effective, is inefficient. The straightforward algorithm yielded by this description involves checking all possible 3-valued instances of a program, determining which are 3-stable models, and then taking their intersection. We next provide a simpler, efficient way of computing the well-founded semantics. It is based on an "alternating fixpoint" computation that converges to the well-founded semantics. As a side-effect, the proof will show that each datalog$^\neg$ program has at least one 3-stable model (and therefore the well-founded semantics is always defined), something we have not proven. It will also show that the well-founded model is itself a 3-stable model, in some sense the smallest.

The idea of the computation is as follows. We define an alternating sequence $\{\mathbf{I}_i\}_{i \geq 0}$ of 3-valued instances that are underestimates and overestimates of the facts known in every

3-stable model of P. The sequence is as follows:

$$\begin{aligned}\mathbf{I}_0 &= \bot \\ \mathbf{I}_{i+1} &= conseq_P(\mathbf{I}_i).\end{aligned}$$

Recall that $\bot$ is the least 3-valued instance and that all facts have value 0 in $\bot$. Also note that each of the $\mathbf{I}_i$ just defined is a total instance. This follows easily from the following facts (Exercise 15.17):

- if $\mathbf{I}$ is total, then $conseq_P(\mathbf{I})$ is total; and
- the $\mathbf{I}_i$ are constructed starting from the total instance $\bot$ by repeated applications of $conseq_P$.

The intuition behind the construction of the sequence $\{\mathbf{I}_i\}_{i\geq 0}$ is the following. The sequence starts with $\bot$, which is an overestimate of the negative facts in the answer (it contains all negative facts). From this overestimate we compute $\mathbf{I}_1 = conseq_P(\bot)$, which includes all positive facts that can be inferred from $\bot$. This is clearly an overestimate of the positive facts in the answer, so the set of negative facts in $\mathbf{I}_1$ is an underestimate of the negative facts in the answer. Using this underestimate of the negative facts, we compute $\mathbf{I}_2 = conseq_P(\mathbf{I}_1)$, whose positive facts will now be an underestimate of the positive facts in the answer. By continuing the process, we see that the even-indexed instances provide underestimates of the positive facts in the answer and the odd-indexed ones provide underestimates of the negative facts in the answer. Then the limit of the even-indexed instances provides the positive facts in the answer and the limit of the odd-indexed instances provides the negative facts in the answer. This intuition will be made formal later in this section.

It is easy to see that $conseq_P(\mathbf{I})$ is antimonotonic. That is, if $\mathbf{I} \prec \mathbf{J}$, then $conseq_P(\mathbf{J}) \prec conseq_P(\mathbf{I})$ (Exercise 15.17). From this and the facts that $\bot \prec \mathbf{I}_1$ and $\bot \prec \mathbf{I}_2$, it immediately follows that, for all $i > 0$,

$$\mathbf{I}_0 \prec \mathbf{I}_2 \ldots \prec \mathbf{I}_{2i} \prec \mathbf{I}_{2i+2} \prec \ldots \prec \mathbf{I}_{2i+1} \prec \mathbf{I}_{2i-1} \prec \ldots \prec \mathbf{I}_1.$$

Thus the even subsequence is increasing and the odd one is decreasing. Because there are finitely many 3-valued instances relative to a given program P, each of these sequences becomes constant at some point. Let $\mathbf{I}_*$ denote the limit of the increasing sequence $\{\mathbf{I}_{2i}\}_{i\geq 0}$, and let $\mathbf{I}^*$ denote the limit of the decreasing sequence $\{\mathbf{I}_{2i+1}\}_{i\geq 0}$. From the aforementioned inequalities, it follows that $\mathbf{I}_* \prec \mathbf{I}^*$. Moreover, note that $conseq_P(\mathbf{I}_*) = \mathbf{I}^*$ and $conseq_P(\mathbf{I}^*) = \mathbf{I}_*$. Finally let $\mathbf{I}^*_*$ denote the 3-valued instance consisting of the facts known in both $\mathbf{I}_*$ and $\mathbf{I}^*$; that is,

$$\mathbf{I}^*_*(A) = \begin{cases} 1 & \text{if } \mathbf{I}_*(A) = \mathbf{I}^*(A) = 1 \\ 0 & \text{if } \mathbf{I}_*(A) = \mathbf{I}^*(A) = 0 \text{ and} \\ 1/2 & \text{otherwise.} \end{cases}$$

Equivalently, $\mathbf{I}^*_* = (\mathbf{I}_*)^1 \cup (\mathbf{I}^*)^0$. As will be seen shortly, $\mathbf{I}^*_* = P^{wf}(\emptyset)$. Before proving this, we illustrate the alternating fixpoint computation with several examples.

EXAMPLE 15.3.8

(a) Consider again the program in Example 15.3.6. Let us perform the alternating fixpoint computation described earlier. We start with $\mathbf{I}_0 = \bot = \{\neg p, \neg q, \neg r, \neg s, \neg t, \neg u\}$. By applying $conseq_P$, we obtain the following sequence of instances:

$$\mathbf{I}_1 = \{p, q, \neg r, s, t, u\},$$
$$\mathbf{I}_2 = \{p, q, \neg r, \neg s, \neg t, \neg u\},$$
$$\mathbf{I}_3 = \{p, q, \neg r, s, t, u\},$$
$$\mathbf{I}_4 = \{p, q, \neg r, \neg s, \neg t, \neg u\}.$$

Thus $\mathbf{I}_* = \mathbf{I}_4 = \{p, q, \neg r, \neg s, \neg t, \neg u\}$ and $\mathbf{I}^* = \mathbf{I}_3 = \{p, q, \neg r, s, t, u\}$. Finally $\mathbf{I}^*_* = \{p, q, \neg r\}$, which coincides with the well-founded semantics of P computed in Example 15.3.6.

(b) Recall now P_{win} and input $\mathbf{K}$ of Example 15.3.1. We compute $\mathbf{I}^*_*$ for the program $P_{win,\mathbf{I}}$. Note that for $\mathbf{I}_0$ the value of all *move* atoms is **false**, and for each $j \geq 1$, $\mathbf{I}_j$ agrees with the input $\mathbf{K}$ on the predicate *moves*; thus we do not show the *move* atoms here. For the *win* predicate, then, we have

$$\mathbf{I}_1 = \{win(a), win(b), win(c), win(d), \neg win(e), win(f), \neg win(g)\}$$
$$\mathbf{I}_2 = \{\neg win(a), \neg win(b), \neg win(c), win(d), \neg win(e), win(f), \neg win(g)\}$$
$$\mathbf{I}_3 = \mathbf{I}_1$$
$$\mathbf{I}_4 = \mathbf{I}_2.$$

Thus

$$\mathbf{I}_* = \mathbf{I}_2 = \{\neg win(a), \neg win(b), \neg win(c), win(d), \neg win(e), win(f), \neg win(g)\}$$
$$\mathbf{I}^* = \mathbf{I}_1 = \{win(a), win(b), win(c), win(d), \neg win(e), win(f), \neg win(g)\}$$
$$\mathbf{I}^*_* = \{win(d), \neg win(e), win(f), \neg win(g)\},$$

which is the instance $\mathbf{J}$ of Example 15.3.1.

(c) Consider the database schema consisting of a binary relation G and a unary relation *good* and the following program defining *bad* and *answer*:

$$bad(x) \leftarrow G(y, x), \neg good(y)$$
$$answer(x) \leftarrow \neg bad(x)$$

Consider the instance $\mathbf{K}$ over G and *good*, where

$$\mathbf{K}(G) = \{\langle b, c\rangle, \langle c, b\rangle, \langle c, d\rangle, \langle a, d\rangle, \langle a, e\rangle\}, \text{ and}$$
$$\mathbf{K}(good) = \{\langle a\rangle\}.$$

We assume that the facts of the database are added as unit clauses to P, yielding $P_{\mathbf{K}}$. Again we perform the alternating fixpoint computation for $P_{\mathbf{K}}$. We start with

$\mathbf{I}_0 = \perp$ (containing all negated atoms). Applying $conseq_{P_\mathbf{K}}$ yields the following sequence $\{\mathbf{I}_i\}_{i>0}$:

	bad	*answer*
$\mathbf{I}_0$	$\emptyset$	$\emptyset$
$\mathbf{I}_1$	$\{\neg a, b, c, d, e\}$	$\{a, b, c, d, e\}$
$\mathbf{I}_2$	$\{\neg a, b, c, d, \neg e\}$	$\{a, \neg b, \neg c, \neg d, \neg e\}$
$\mathbf{I}_3$	$\{\neg a, b, c, d, \neg e\}$	$\{a, \neg b, \neg c, \neg d, e\}$
$\mathbf{I}_4$	$\{\neg a, b, c, d, \neg e\}$	$\{a, \neg b, \neg c, \neg d, e\}$

We have omitted [as in (b)] the facts relating to the *edb* predicates G and *good*, which do not change after step 1.

Thus $\mathbf{I}^*_* = \mathbf{I}_* = \mathbf{I}^* = \mathbf{I}_3 = \mathbf{I}_4$. Note that P is stratified and its well-founded semantics coincides with its stratified semantics. As we shall see, this is not accidental.

We now show that the fixpoint construction yields the well-founded semantics for datalog$^\neg$ programs.

THEOREM 15.3.9 For each datalog$^\neg$ program P,

1. $\mathbf{I}^*_*$ is a 3-stable model of P.
2. $P^{wf}(\emptyset) = \mathbf{I}^*_*$.

Proof For statement 1, we need to show that $conseq_P(\mathbf{I}^*_*) = \mathbf{I}^*_*$. We show that for every fact A, if $\mathbf{I}^*_*(A) = \epsilon \in \{0, 1/2, 1\}$, then $conseq_P(\mathbf{I}^*_*)(A) = \epsilon$. From the antimonotonicity of $conseq_P$, the fact that $\mathbf{I}_* \prec \mathbf{I}^*_* \prec \mathbf{I}^*$ and $conseq_P(\mathbf{I}_*) = \mathbf{I}^*$, $conseq_P(\mathbf{I}^*) = \mathbf{I}_*$, it follows that $\mathbf{I}_* \prec conseq_P(\mathbf{I}^*_*) \prec \mathbf{I}^*$. If $\mathbf{I}^*_*(A) = 0$, then $\mathbf{I}^*(A) = 0$ so $conseq_P(\mathbf{I}^*_*)(A) = 0$; similarly for $\mathbf{I}^*_*(A) = 1$. Now suppose that $\mathbf{I}^*_*(A) = 1/2$. It is sufficient to prove that $conseq_P(\mathbf{I}^*_*)(A) \geq 1/2$. [It is not possible that $conseq_P(\mathbf{I}^*_*)(A) = 1$. If this were the case, the rules used to infer A involve only facts whose value is 0 or 1. Because those facts have the same value in $\mathbf{I}_*$ and $\mathbf{I}^*$, the same rules can be used in both $pg(P, \mathbf{I}_*)$ and $pg(P, \mathbf{I}^*)$ to infer A, so $\mathbf{I}_*(A) = \mathbf{I}^*(A) = \mathbf{I}^*_*(A) = 1$, which contradicts the hypothesis that $\mathbf{I}^*_*(A) = 1/2$.]

We now prove that $conseq_P(\mathbf{I}^*_*)(A) \geq 1/2$. By the definition of $\mathbf{I}^*_*$, $\mathbf{I}_*(A) = 0$ and $\mathbf{I}^*(A) = 1$. Recall that $conseq_P(\mathbf{I}_*) = \mathbf{I}^*$, so $conseq_P(\mathbf{I}_*)(A) = 1$. In addition, $conseq_P(\mathbf{I}_*)$ is the limit of the sequence $\{3\text{-}T^i_{pg(P,\mathbf{I}_*)}\}_{i>0}$. Let $stage(A)$ be the minimum i such that $3\text{-}T^i_{pg(P,\mathbf{I}_*)}(A) = 1$. We prove by induction on $stage(A)$ that $conseq_P(\mathbf{I}^*_*)(A) \geq 1/2$. Suppose that $stage(A) = 1$. Then there exists in $ground(P)$ a rule of the form $A \leftarrow$, or one of the form $A \leftarrow \neg B_1, \ldots, \neg B_n$, where $\mathbf{I}_*(B_j) = 0, 1 \leq j \leq n$. However, the first case cannot occur, for otherwise $conseq_P(\mathbf{I}^*)(A)$ must also equal 1 so $\mathbf{I}_*(A) = 1$ and therefore $\mathbf{I}^*_*(A) = 1$, contradicting the fact that $\mathbf{I}^*_*(A) = 1/2$. By the same argument, $\mathbf{I}^*(B_j) = 1$, so $\mathbf{I}^*_*(B_j) = 1/2, 1 \leq j \leq n$. Consider now $pg(P, \mathbf{I}^*_*)$. Because $\mathbf{I}^*_*(B_j) =$

$1/2$, $1 \leq j \leq n$, the second rule yields $conseq_P(\mathbf{I}^*_*)(A) \geq 1/2$. Now suppose that the statement is true for $stage(A) = i$ and suppose that $stage(A) = i + 1$. Then there exists a rule $A \leftarrow A_1 \ldots A_m \neg B_1 \ldots \neg B_n$ such that $\mathbf{I}_*(B_j) = 0$ and $3\text{-}T^i_{pg(P,\mathbf{I}_*)}(A_k) = 1$ for each j and k. Because $\mathbf{I}_*(B_j) = 0$, $\mathbf{I}^*_*(B_j) \leq 1/2$ so $\mathbf{I}^*_*(\neg B_j) \geq 1/2$. In addition, by the induction hypothesis, $conseq_P(\mathbf{I}^*_*)(A_k) \geq 1/2$. It follows that $conseq_P(\mathbf{I}^*_*)(A) \geq 1/2$, and the induction is complete. Thus $conseq_P(\mathbf{I}^*_*) = \mathbf{I}^*_*$ and $\mathbf{I}^*_*$ is a 3-stable model of P.

Consider statement 2. We have to show that the positive and negative facts in $\mathbf{I}^*_*$ are those belonging to every 3-stable model $\mathbf{M}$ of P. Because $\mathbf{I}^*_*$ is itself a 3-stable model of P, it contains the positive and negative facts belonging to every 3-stable model of P. It remains to show the converse (i.e., that the positive and negative facts in $\mathbf{I}^*_*$ belong to every 3-stable model of P). To this end, we first show that for each 3-stable model $\mathbf{M}$ of P and $i \geq 0$,

(‡) $$\mathbf{I}_{2i} \prec \mathbf{M} \prec \mathbf{I}_{2i+1}.$$

The proof is by induction on i. For $i = 0$, we have

$$\mathbf{I}_0 = \bot \prec \mathbf{M}.$$

Because $conseq_P$ is antimonotonic, $conseq_P(\mathbf{M}) \prec conseq_P(\mathbf{I}_0)$. Now $conseq_P(\mathbf{I}_0) = \mathbf{I}_1$ and because $\mathbf{M}$ is 3-stable, $conseq_P(\mathbf{M}) = \mathbf{M}$. Thus we have

$$\mathbf{I}_0 \prec \mathbf{M} \prec \mathbf{I}_1.$$

The induction step is similar and is omitted.

By (‡), $\mathbf{I}_* \prec \mathbf{M} \prec \mathbf{I}^*$. Now a positive fact in $\mathbf{I}^*_*$ is in $\mathbf{I}_*$ and so is in $\mathbf{M}$ because $\mathbf{I}_* \prec \mathbf{M}$. Similarly, a negative fact in $\mathbf{I}^*_*$ is in $\mathbf{I}^*$ and so is in $\mathbf{M}$ because $\mathbf{M} \prec \mathbf{I}^*$. ■

Note that the proof of statement 2 above formalizes the intuition that the $\mathbf{I}_{2i}$ provide underestimates of the positive facts in all acceptable answers (3-stable models) and the $\mathbf{I}_{2i+1}$ provide underestimates of the negative facts in those answers. The fact that $P^{wf}(\emptyset)$ is a minimal model of P is left for Exercise 15.19.

Variations of the alternating fixpoint computation can be obtained by starting with initial instances different from $\bot$. For example, it may make sense to start with the content of the *edb* relations as an initial instance. Such variations are sometimes useful for technical reasons. It turns out that the resulting sequences still compute the well-founded semantics. We show the following:

PROPOSITION 15.3.10 Let P be a datalog$^\neg$ program. Let $\{\bar{\mathbf{I}}_i\}_{i \geq 0}$ be defined in the same way as the sequence $\{\mathbf{I}_i\}_{i \geq 0}$, except that $\bar{\mathbf{I}}_0$ is some total instance such that

$$\bot \prec \bar{\mathbf{I}}_0 \prec P^{wf}(\emptyset).$$

Then

$$\bar{\mathbf{I}}_0 \prec \bar{\mathbf{I}}_2 \ldots \prec \bar{\mathbf{I}}_{2i} \prec \bar{\mathbf{I}}_{2i+2} \prec \ldots \prec \bar{\mathbf{I}}_{2i+1} \prec \bar{\mathbf{I}}_{2i-1} \prec \ldots \prec \bar{\mathbf{I}}_1$$

and (using the same notation as before),

$$\bar{\mathbf{I}}^*_* = P^{wf}(\emptyset).$$

Proof Let us compare the sequences $\{\mathbf{I}_i\}_{i\geq 0}$ and $\{\bar{\mathbf{I}}_i\}_{i\geq 0}$. Because $\bar{\mathbf{I}}_0 \prec P^{wf}(\emptyset)$ and $\bar{\mathbf{I}}_0$ is total, it easily follows that $\bar{\mathbf{I}}_0 \prec \mathbf{I}_*$. Thus $\perp = \mathbf{I}_0 \prec \bar{\mathbf{I}}_0 \prec \mathbf{I}_*$. From the antimonotonicity of the $conseq_P$ operator and the fact that $conseq^2_P(\mathbf{I}_*) = \mathbf{I}_*$, it follows that $\mathbf{I}_{2i} \prec \bar{\mathbf{I}}_{2i} \prec \mathbf{I}_*$ for all i, $i \geq 0$. Thus $\bar{\mathbf{I}}_* = \mathbf{I}_*$. Then

$$\bar{\mathbf{I}}^* = conseq_P(\bar{\mathbf{I}}_*) = conseq_P(\mathbf{I}_*) = \mathbf{I}^*$$

so $\bar{\mathbf{I}}^*_* = \mathbf{I}^*_* = P^{wf}(\emptyset)$. ■

As noted earlier, the instances in the sequence $\{\mathbf{I}_i\}_{i\geq 0}$ are total. A slightly different alternating fixpoint computation formulated only in terms of positive and negative facts can be defined. This is explored in Exercise 15.25.

Finally, the alternating fixpoint computation of the well-founded semantics involves looking at the ground rules of the given program. However, one can clearly compute the semantics without having to explicitly look at the ground rules. We show in Section 15.4 how the well-founded semantics can be computed by a *fixpoint* query.

Well-Founded and Stratified Semantics Agree

Because the well-founded semantics provides semantics to all datalog$^\neg$ programs, it does so in particular for stratified programs. Example 15.3.8(c) showed one stratified program for which stratified and well-founded semantics coincide. Fortunately, as shown next, stratified and well-founded semantics are always compatible. Thus if a program is stratified, then the stratified and well-founded semantics agree.

A datalog$^\neg$ program P is said to be *total* if $P^{wf}(\mathbf{I})$ is total for each input $\mathbf{I}$ over $edb(P)$.

Theorem 15.3.11 If P is a stratified datalog$^\neg$ program, then P is total under the well-founded semantics, and for each 2-valued instance $\mathbf{I}$ over $edb(P)$, $P^{wf}(\mathbf{I}) = P^{strat}(\mathbf{I})$.

Proof Let P be stratified, and let input $\mathbf{I}_0$ over $edb(P)$ be fixed. The idea of the proof is the following. Let $\mathbf{J}$ be a 3-stable model of $P_{\mathbf{I}_0}$. We shall show that $\mathbf{J} = P^{strat}(\mathbf{I}_0)$. This will imply that $P^{strat}(\mathbf{I}_0)$ is the unique 3-stable model for $P_{\mathbf{I}_0}$. In particular, it contains only the positive and negative facts in all 3-stable models of $P_{\mathbf{I}_0}$ and is thus $P^{wf}(\mathbf{I}_0)$.

For the proof, we will need to develop some notation.

Notation for the stratification: Let $P^1, \ldots, P^n$ be a stratification of P. Let $P^0 = \emptyset_{\mathbf{I}_0}$ (i.e., the program corresponding to all of the facts in $\mathbf{I}_0$). For each k in $[0, n]$,

- let $\mathbf{S}_k = idb(P^k)$ ($\mathbf{S}_0$ is $edb(P)$);
- $\mathbf{S}_{[0,k]} = \cup_{i\in[0,k]}\mathbf{S}_i$; and

▪ $\mathbf{I}_k = (P^1 \cup \cdots \cup P^k)^{strat}(\mathbf{I}_0) = \mathbf{I}_n|\mathbf{S}_{[0,k]}$ (and, in particular, $P^{strat}(\mathbf{I}_0) = \mathbf{I}_n$).

Notation for the 3-stable model: Let $\hat{P} = pg(P_{\mathbf{I}_0}, \mathbf{J})$. Recall that because $\mathbf{J}$ is 3-stable for $P_{\mathbf{I}_0}$,

$$\mathbf{J} = conseq_{\hat{P}}(\mathbf{J}) = \lim_{i \geq 0} 3\text{-}T^i_{\hat{P}}(\emptyset).$$

For each k in $[0, n]$,

- let $\mathbf{J}_k = \mathbf{J}|\mathbf{S}_{[0,k]}$; and
- $\hat{P}^{k+1} = pg(P^{k+1}_{\mathbf{J}_k}, \mathbf{J}_k) = pg(P^{k+1}_{\mathbf{J}_k}, \mathbf{J})$.

[Note that $pg(P^{k+1}_{\mathbf{J}_k}, \mathbf{J}_k) = pg(P^{k+1}_{\mathbf{J}_k}, \mathbf{J})$ because all the negations in P^{k+1} are over predicates in $\mathbf{S}_{[0,k]}$.]

To demonstrate the result, we will show by induction on $k \in [0, n]$ that

(*) $$\exists l_k \geq 0 \text{ such that } \forall i \geq 0, \mathbf{J}_k = 3\text{-}T^{l_k+i}_{\hat{P}}(\emptyset) \mid \mathbf{S}_{[0,k]} = \mathbf{I}_k.$$

Clearly, for $k = n$, (*) demonstrates the result.

The case where $k = 0$ is satisfied by setting $l_0 = 1$, because $\mathbf{J}_0 = 3\text{-}T^{1+i}_{\hat{P}}(\emptyset)|\mathbf{S}_0 = \mathbf{I}_0$ for each $i \geq 0$.

Suppose now that (*) is true for some $k \in [0, n-1]$. Then for each $i \geq 0$, by the choice of $\hat{P}^{k+1}$, the form of P^{k+1}, and (*),

(1) $$T^i_{P^{k+1}}(\mathbf{I}_k)|\mathbf{S}_{k+1} \subseteq 3\text{-}T^{i+1}_{\hat{P}^{k+1}}(\emptyset)|\mathbf{S}_{k+1} \subseteq T^{i+1}_{P^{k+1}}(\mathbf{I}_k)|\mathbf{S}_{k+1}.$$

(Here and later, $\subseteq$ denotes the usual 2-valued containment between instances; this is well defined because all instances considered are total, even if $\mathbf{J}$ is not.) In (1), the $3\text{-}T^{i+1}_{\hat{P}^{k+1}}$ and $T^{i+1}_{P^{k+1}}$ terms may not be equal, because the positive atoms of $\mathbf{I}_k = \mathbf{J}_k$ are available when applying $T_{P^{k+1}}$ the first time but are available only during the second application of $3\text{-}T_{\hat{P}^{k+1}}$. On the other hand, the $T^i_{P^{k+1}}$ and $3\text{-}T^{i+1}_{\hat{P}^{k+1}}$ terms may not be equal (e.g., if there is a rule of the form $A \leftarrow$ in P^{k+1}).

By (1) and finiteness of the input, there is some $m \geq 0$ such that for each $i \geq 0$,

(2) $$\mathbf{I}_n|\mathbf{S}_{k+1} = T^{m+i}_{P^{k+1}}(\mathbf{I}_k)|\mathbf{S}_{k+1} = 3\text{-}T^{m+i}_{\hat{P}^{k+1}}(\emptyset)|\mathbf{S}_{k+1}.$$

This is almost what is needed to complete the induction, except that $\hat{P}^{k+1}$ is used instead of $\hat{P}$. However, observe that for each $i \geq 0$,

(3) $$3\text{-}T^i_{\hat{P}}(\emptyset)|\mathbf{S}_{k+1} \subseteq 3\text{-}T^i_{\hat{P}^{k+1}}(\emptyset)|\mathbf{S}_{k+1}$$

because $3\text{-}T^i_{\hat{P}}(\emptyset)|\mathbf{S}_{[0,k]} \subseteq \mathbf{J}_k$ for each $i \geq 0$ by the induction hypothesis. Finally observe that for each $i \geq 0$,

(4) $$3\text{-}T^i_{\hat{P}^{k+1}}(\emptyset)|\mathbf{S}_{k+1} \subseteq 3\text{-}T^{i+l_k}_{\hat{P}}(\emptyset)|\mathbf{S}_{k+1}$$

because $3\text{-}T_{\hat{P}}^{l_k}(\emptyset)|\mathbf{S}_{[0,k]}$ contains all of the positive atoms of $\mathbf{J}_k$.

Then for each $i \geq 0$ we have

$$\begin{aligned} 3\text{-}T_{\hat{P}^{k+1}}^{m+i}(\emptyset)|\mathbf{S}_{k+1} &\subseteq 3\text{-}T_{\hat{P}}^{m+i+l_k}(\emptyset)|\mathbf{S}_{k+1} && \text{by (4)} \\ &\subseteq 3\text{-}T_{\hat{P}^{k+1}}^{m+i+l_k}(\emptyset)|\mathbf{S}_{k+1} && \text{by (3)} \\ &\subseteq 3\text{-}T_{\hat{P}^{k+1}}^{m+i}(\emptyset)|\mathbf{S}_{k+1} && \text{by (2).} \end{aligned}$$

It follows that

$$3\text{-}T_{\hat{P}^{k+1}}^{m+i}(\emptyset)|\mathbf{S}_{k+1} = 3\text{-}T_{\hat{P}}^{m+i+l_k}(\emptyset)|\mathbf{S}_{k+1}. \tag{5}$$

Set $l_{(k+1)} = l_k + m$. Combining (2) and (5), we have, for each $i \geq 0$,

$$\mathbf{J}|\mathbf{S}_{k+1} = 3\text{-}T_{\hat{P}}^{l_{(k+1)}+i}(\emptyset)|\mathbf{S}_{k+1} = \mathbf{I}_n|\mathbf{S}_{k+1}.$$

Together with the inductive hypothesis, we obtain for each $i \geq 0$ that

$$\mathbf{J}|\mathbf{S}_{[0,k+1]} = 3\text{-}T_{\hat{P}}^{l_{(k+1)}+i}(\emptyset)|\mathbf{S}_{[0,k+1]} = \mathbf{I}_n|\mathbf{S}_{[0,k+1]},$$

which concludes the proof. ■

As just seen, each stratifiable program is total under the well-founded semantics. However, as indicated by Example 15.3.8(b), a datalog$^\neg$ program P may yield a 3-valued model $P^{wf}(\mathbf{I})$ on some inputs. Furthermore, there are programs that are not stratified but whose well-founded models are nonetheless total (see Exercise 15.22). Unfortunately, there can be no effective characterization of those datalog$^\neg$ programs whose well-founded semantics is total for all input databases (Exercise 15.23). One can find sufficient syntactic conditions that guarantee the totality of the well-founded semantics, but this quickly becomes a tedious endeavor. It has been shown, however, that for each datalog$^\neg$ program P, one can find another program whose well-founded semantics is total on all inputs and that produces the same positive facts as the well-founded semantics of P.

15.4 Expressive Power

In this section, we examine the expressive power of datalog$^\neg$ with the various semantics for negation we have considered. More precisely, we focus on semipositive, stratified, and well-founded semantics. We first look at the relative power of these semantics and show that semipositive programs are weaker than stratified, which in turn are weaker than well founded. Then we look at the connection with languages studied in Chapter 14 that also use recursion and negation. We prove that well-founded semantics can express precisely the *fixpoint* queries.

Finally we look at the impact of *order* on expressive power. An ordered database contains a special binary relation *succ* that provides a successor relation on all constants

in the active domain. Thus the constants are ordered by *succ* and in fact can be viewed as integers. The impact of assuming that a database is ordered is examined at length in Chapter 17. Rather surprisingly, we show that in the presence of order, semipositive programs are as powerful as programs with well-founded semantics. In particular, all three semantics are equivalent and express precisely the *fixpoint* queries.

We begin by briefly noting the connection between stratified datalog$^\neg$ and relational calculus (and algebra). To see that stratified datalog$^\neg$ can express all queries in CALC, recall the nonrecursive datalog$^\neg$ (nr-datalog$^\neg$) programs introduced in Chapter 5. Clearly, these are stratified datalog$^\neg$ programs in which recursion is not allowed. Theorem 5.3.10 states that nr-datalog$^\neg$ (with one answer relation) and CALC are equivalent. It follows that stratified datalog$^\neg$ can express all of CALC. Because transitive closure of a graph can be expressed in stratified datalog$^\neg$ but not in CALC (see Proposition 17.2.3), it follows that stratified datalog$^\neg$ is strictly stronger than CALC.

Stratified Datalog Is Weaker than *Fixpoint*

Let us look at the expressive power of stratified datalog$^\neg$. Computationally, stratified programs provide recursion and negation and are inflationary. Therefore one might expect that they express the *fixpoint* queries. It is easy to see that all stratified datalog$^\neg$ are *fixpoint* queries (Exercise 15.28). In particular, this shows that such programs can be evaluated in polynomial time. Can stratified datalog$^\neg$ express all *fixpoint* queries? Unfortunately, no. The intuitive reason is that in stratified datalog$^\neg$ there is no recursion through negation, so the number of applications of negation is bounded. In contrast, *fixpoint* queries allow recursion through negation, so there is no bound on the number of applications of negation. This distinction turns out to be crucial. We next outline the main points of the argument, showing that stratified datalog$^\neg$ is indeed strictly weaker than *fixpoint*.

The proof uses a game played on so-called game trees. The game is played on a given tree. The nodes of the tree are the possible positions in the game, and the edges are the possible moves from one position to another. Additionally, some leaves of the tree are labeled black. The game is between two players. A round of the game starting at node x begins with Player I making a move from x to one of its children y. Player II then makes a move from y, etc. The game ends when a leaf is reached. Player I wins if Player II picks a black leaf. For a given tree (with labels), Player I has a winning strategy for the game starting at node x if he or she can win starting at x no matter how Player II plays. We are interested in programs determining whether there is such a winning strategy.

The game tree is represented as follows. The set of possible moves is given by a binary relation *move* and the set of black nodes by a unary relation *black*. Consider the query *winning* (not to be confused with the predicate *win* of Example 15.3.1), which asks if Player I has a winning strategy starting at the root of the tree. We will define a set of game trees $\mathcal{G}$ such that

(i) the query *winning* on the game trees in $\mathcal{G}$ is definable by a *fixpoint* query, and

(ii) for each stratified program P, there exist game trees $G, G' \in \mathcal{G}$ such that *winning* is true on G and false on G', but P cannot distinguish between G and G'.

Clearly, (ii) shows that the *winning* query on game trees is not definable by a stratified

datalog$^\neg$ program. The set $\mathcal{G}$ of game trees is defined next. It consists of the $G_{l,k}$ and $G'_{l,k}$ defined by induction as follows:

- $G_{0,k}$ and $G'_{0,k}$ have no moves and just one node, labeled black in $G_{0,k}$ and not labeled in $G'_{0,k}$.
- $G_{i+1,k}$ consists of a copy of $G'_{i,k}$, k disjoint copies of $G_{i,k}$, and a new root d_{i+1}. The moves are the union of the moves in the copies of $G'_{i,k}$ and $G_{i,k}$ together with new moves from the root d_{i+1} to the roots of the copies. The labels remain unchanged.
- $G'_{i+1,k}$ consists of $k+1$ disjoint copies of $G_{i,k}$ and a new root d'_{i+1} from which moves are possible to the roots of the copies of $G_{i,k}$.

The game trees $G_{4,1}$ and $G'_{4,1}$ are represented in Fig. 15.2. It is easy to see that *winning* is true on the game trees $G_{2i,k}$ and false on game trees $G'_{2i,k}$, $i > 0$ (Exercise 15.30).

We first note that the query *winning* on game trees in $\mathcal{G}$ can be defined by a *fixpoint* query. Consider

$$\begin{aligned}\varphi(T) = {} & (\exists y)[Move(x, y) \wedge (\forall z)(Move(y, z) \rightarrow Black(z))] \\ & \vee (\exists y)[Move(x, y) \wedge (\forall z)(Move(y, z) \rightarrow T(z))].\end{aligned}$$

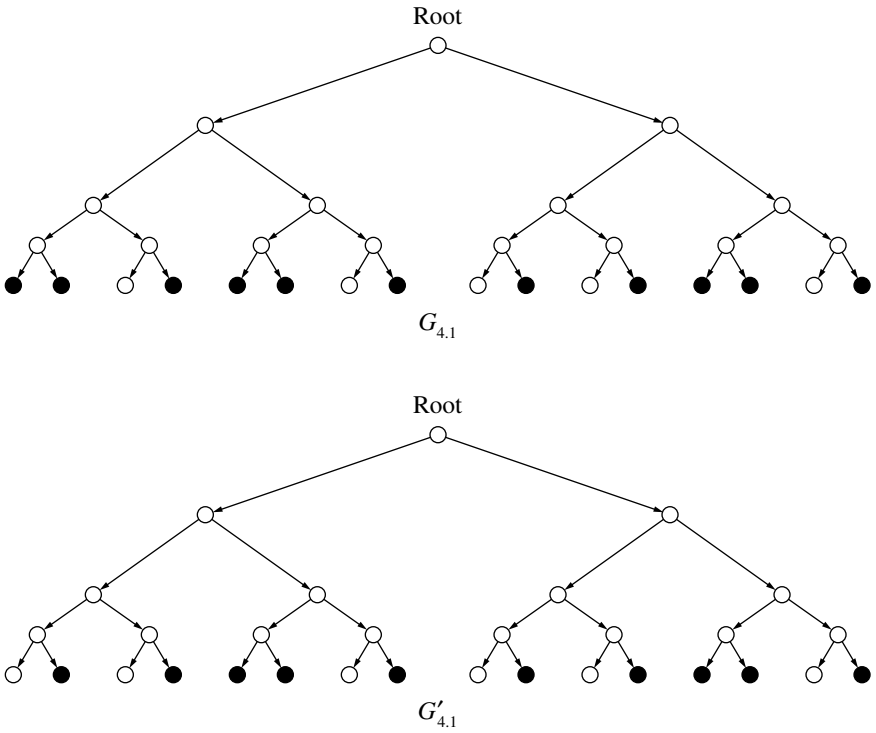


Figure 15.2: Game trees

It is easy to verify that *winning* is defined by $\mu_T(\varphi(T))(root)$, where *root* is the root of the game tree (Exercise 15.30). Next we note that the *winning* query is not expressible by any stratified datalog$^\neg$ program. To this end, we use the following result, stated without proof.

LEMMA 15.4.1 For each stratified datalog$^\neg$ program P, there exist i, k such that

$$P(G_{i,k})(winning) = P(G'_{i,k})(winning).$$

The proof of Lemma 15.4.1 uses an extension of Ehrefeucht-Fraissé games (the games are described in Chapter 17). The intuition of the lemma is that, to distinguish between $G_{i,k}$ and $G'_{i,k}$ for i and k sufficiently large, one needs to apply more negations than the fixed number allowed by P. Thus no stratified program can distinguish between all the $G_{i,k}$ and $G'_{i,k}$. In particular, it follows that the *fixpoint* query *winning* is not equivalent to *any* stratified datalog$^\neg$ program. Thus we have the following result, settling the relationship between stratified datalog$^\neg$ and the *fixpoint* queries.

THEOREM 15.4.2 The class of queries expressible by stratified datalog$^\neg$ programs is strictly included in the *fixpoint* queries.

REMARK 15.4.3 The game tree technique can also be used to prove that the number of strata in stratified datalog$^\neg$ programs has an impact on expressive power. Specifically, let $Strat_i$ consist of all queries expressible by stratified datalog$^\neg$ programs with i strata. Then it can be shown that for all i, $Strat_i \subset Strat_{i+1}$. In particular, semipositive datalog$^\neg$ is weaker than stratified datalog$^\neg$.

Well-Founded Datalog$^\neg$ Is Equivalent to *Fixpoint*

Next we consider the expressive power of datalog$^\neg$ programs with well-founded semantics. We prove that well-founded semantics can express precisely the *fixpoint* queries. We begin by showing that the well-founded semantics can be computed by a *fixpoint* query. More precisely, we show how to compute the set of false, true, and undefined facts of the answer using a *while*$^+$ program (see Chapter 14 for the definition of *while*$^+$ programs).

THEOREM 15.4.4 Let P be a datalog$^\neg$ program. There exists a *while*$^+$ program w with input relations $edb(P)$, such that

1. w contains, for each relation R in $sch(P)$, three relation variables R^ϵ_{answer}, where $\epsilon \in \{0, 1/2, 1\}$;
2. for each instance $\mathbf{I}$ over $edb(P)$, $u \in w(\mathbf{I})(R^\epsilon_{answer})$ iff $P^{wf}(\mathbf{I})(R(u)) = \epsilon$, for $\epsilon \in \{0, 1/2, 1\}$.

Crux Let P be a datalog$^\neg$ program. The *while*$^+$ program mimics the alternating fixpoint computation of P^{wf}. Recall that this involves repeated applications of the operator $conseq_P$, resulting in the sequence

$$\mathbf{I}_0 \prec \mathbf{I}_2 \ldots \prec \mathbf{I}_{2i} \prec \mathbf{I}_{2i+2} \prec \ldots \prec \mathbf{I}_{2i+1} \prec \mathbf{I}_{2i-1} \prec \ldots \prec \mathbf{I}_1.$$

Recall that the $\mathbf{I}_i$ are all total instances. Thus 3-valued instances are only required to produce the final answer from $\mathbf{I}_*$ and $\mathbf{I}^*$ at the end of the computation, by one last first-order query.

It is easily verified that *while*$^+$ can simulate one application of $conseq_P$ on total instances (Exercise 15.27). The only delicate point is to make sure the computation is inflationary. To this end, the program w will distinguish between results of even and odd iterations of $conseq_P$ by having, for each R, an odd and even version R^0_{odd} and R^1_{even}. R^0_{odd} holds at iteration $2i+1$ the negative facts of R in $\mathbf{I}_{2i+1}$, and R^1_{even} holds at iteration $2i$ the positive facts of R in $\mathbf{I}_{2i}$. Note that both R^0_{odd} and R^1_{even} are increasing throughout the computation.

We elaborate on the simulation of the operator $conseq_P$ on a total instance $\mathbf{I}$. The program w will have to distinguish between facts in the input $\mathbf{I}$, used to resolve the negative premises of rules in P, and those inferred by applications of 3-T_P. Therefore for each relation R, the *while*$^+$ program will also maintain a copy $\bar{R}_{even}$ and $\bar{R}_{odd}$ to hold the facts produced by consecutive applications of 3-T_P in the even and odd cases, respectively. More precisely, the $\bar{R}_{odd}$ hold the positive facts inferred from input $\mathbf{I}_{2i}$ represented in R^1_{even}, and the $\bar{R}_{even}$ hold the positive facts inferred from input $\mathbf{I}_{2i+1}$ represented in R^0_{odd}. It is easy to write a first-order query defining one application of 3-T_P for the even or odd cases. Because the representations of the input are different in the even and odd cases, different programs must be used in the two cases. This can be iterated in an inflationary manner, because the set of positive facts inferred in consecutive applications of 3-T_P is always increasing. However, the $\bar{R}_{odd}$ and $\bar{R}_{even}$ have to be initialized to $\emptyset$ at each application of $conseq_P$. Because the computation must be inflationary, this cannot be done directly. Instead, timestamping must be used. The initialization of the $\bar{R}_{odd}$ and $\bar{R}_{even}$ is simulated by timestamping each relation with the current content of R^1_{even} and R^0_{odd}, respectively. This is done in a manner similar to the proofs of Chapter 14. ■

We now exhibit a converse of Theorem 15.4.4, showing that any *fixpoint* query can essentially be simulated by a datalog$^\neg$ program with well-founded semantics. More precisely, the positive portion of the well-founded semantics yields the same facts as the *fixpoint* query.

Example 15.4.6 illustrates the proof of this result.

THEOREM 15.4.5 Let q be a *fixpoint* query over input schema $\mathbf{R}$. There exists a datalog$^\neg$ program P such that $edb(P) = \mathbf{R}$, P has an *idb* relation *answer*, and for each instance $\mathbf{I}$ over $\mathbf{R}$, the positive portion of *answer* in $P^{wf}(\mathbf{I})$ coincides with $q(\mathbf{I})$.

Crux We will use the definition of *fixpoint* queries by iterations of positive first-order formulas. Let q be a *fixpoint* query. As discussed in Chapter 14, there exists a CALC formula $\varphi(T)$, positive in T, such that q is defined by $\mu_T(\varphi(T))(u)$, where u is a vector of variables and constants. Consider the CALC formula $\varphi(T)$. As noted earlier in this section, there is an nr-datalog$^\neg$ program P_φ with one answer relation R' such that P_φ is equivalent

to $\varphi(T)$. Because $\varphi(T)$ is positive in T, along any path in the syntax tree of $\varphi(T)$ ending with atom T there is an even number of negations. This is also true of paths in G_{P_φ}.

Consider the precedence graph G_{P_φ} of P_φ. Clearly, one can construct P_φ such that each *idb* relation except T is used in the definition of exactly one other *idb* relation, and all idb relations are used eventually in the definition of the answer R'. In other words, for each *idb* relation R other than T, there is a unique path in G_{P_φ} from R to R'. Consider the paths from T to some *idb* relation R in P_φ. Without loss of generality, we can assume that all paths have the same number of negations (otherwise, because all paths to T have an even number of negations, additional *idb* relations can be introduced to pad the paths with fewer negations, using rules that perform redundant double negations). Let the *rank* of an *idb* relation R in P_φ be the number of negations on each path leading from T to R in G_{P_φ}. Now let P be the datalog$^\neg$ program obtained from P_φ as follows:

- replace the answer relation R' by T;
- add one rule $answer(v) \leftarrow T(u)$, where v is the vector of distinct variables occurring in u, in order of occurrence.

The purpose of replacing R' by T is to cause program P_φ to iterate, yielding $\mu_T(\varphi(T))$. The last rule is added to perform the final selection and projection needed to obtain the answer $\mu_T(\varphi(T))(u)$. Note that, in some sense, P is almost stratified, except for the fact that the result T is fed back into the program.

Consider the alternating fixpoint sequence $\{\mathbf{I}_i\}_{i\geq 0}$ in the computation of $P^{wf}(\mathbf{I})$. Suppose R' has rank q in P_φ, and let R be an *idb* relation of P_φ whose rank in P_φ is $r \leq q$. Intuitively, there is a close correspondence between the sequence $\{\mathbf{I}_i\}_{i\geq 0}$ and the iterations of φ, along the following lines: Each application of $conseq_P$ propagates the correct result from relations of rank r in P_φ to relations of rank $r+1$. There is one minor glitch, however: In the fixpoint computation, the *edb* relations are given, and even at the first iteration, their negation is taken to be their complement; in the alternating fixpoint computation, all negative literals, including those involving *edb* relations, are initially taken to be true. This results in a mismatch. To fix the problem, consider a variation of the alternating fixpoint computation of $P^{wf}(\mathbf{I})$ defined as follows:

$$\begin{aligned}\bar{\mathbf{I}}_0 &= \mathbf{I} \cup \neg.\{R(a_1,\ldots,a_n) \mid R \in idb(P),\ R(a_1,\ldots,a_n) \in \mathbf{B}(P,\mathbf{I})\}\\ \bar{\mathbf{I}}_{i+1} &= conseq_P(\bar{\mathbf{I}}_i).\end{aligned}$$

Clearly, $\perp \prec \bar{\mathbf{I}}_0 \prec P^{wf}(\mathbf{I})$. Then, by Proposition 15.3.10, $\bar{\mathbf{I}}^*_* = P^{wf}(\mathbf{I})$.

Now the following can be verified by induction for each *idb* relation R of rank r:

For each i, $(\bar{\mathbf{I}}_{iq+r})^1$ contains exactly the facts of R true in $P_\varphi(\varphi^i(\emptyset))$.

Intuitively, this is so because each application of $conseq_P$ propagates the correct result across one application of negation to an *idb* predicate. Because R' has rank q, it takes q applications to simulate a complete application of P_φ. In particular, it follows that for each i, $(\bar{\mathbf{I}}_{iq})^1$ contains in T the facts true in $\varphi^i(\emptyset)$.

Thus $(\bar{\mathbf{I}}_*)^1$ contains in T the facts true in $\mu_T(\varphi(T))$. Finally *answer* is obtained by a simple selection and projection from T using the last rule in P and yields $\mu_T(\varphi(T))(u)$. ■

In the preceding theorem, the *positive* portion of *answer* for $P^{wf}(\mathbf{I})$ coincides with $q(\mathbf{I})$. However, $P^{wf}(\mathbf{I})$ is not guaranteed to be total (i.e., it may contain unknown facts). Using a recent result (not demonstrated here), a program Q can be found such that Q^{wf} always provides a total answer, and such that the positive facts of P^{wf} and Q^{wf} coincide on all inputs.

Recall from Chapter 14 that datalog$^\neg$ with inflationary semantics also expresses precisely the *fixpoint* queries. Thus we have converged again, this time by the deductive database path, to the *fixpoint* queries. This bears witness, once more, to the naturalness of this class. In particular, the well-founded and inflationary semantics, although very different, have the same expressive power (modulo the difference between 3-valued and 2-valued models).

Example 15.4.6 Consider the *fixpoint* query $\mu_{good}(\varphi(good))(x)$, where

$$\varphi(good) = \forall y(G(y, x) \rightarrow good(y)).$$

Recall that this query, also encountered in Chapter 14, computes the "good" nodes of the graph G (i.e., those that cannot be reached from a cycle). The nr-datalog$^\neg$ program P_φ corresponding to one application of $\varphi(good)$ is the one exhibited in Example 15.3.8(c):

$$\begin{aligned} bad(x) &\leftarrow G(y, x), \neg good(y) \\ R'(x) &\leftarrow \neg bad(x) \end{aligned}$$

Note that *bad* is negative in P_φ and has rank one, and *good* is positive. The answer R' has rank two. The program P is as follows:

$$\begin{aligned} bad(x) &\leftarrow G(y, x), \neg good(y) \\ good(x) &\leftarrow \neg bad(x) \\ answer(x) &\leftarrow good(x) \end{aligned}$$

Consider the input graph

$$G = \{\langle b, c\rangle, \langle c, b\rangle, \langle c, d\rangle, \langle a, d\rangle, \langle a, e\rangle\}.$$

The consecutive values of $\varphi^i(\emptyset)$ are

$$\begin{aligned} \varphi(\emptyset) &= \{a\}, \\ \varphi^2(\emptyset) &= \{a, e\}, \\ \varphi^3(\emptyset) &= \{a, e\}. \end{aligned}$$

Thus $\mu_{good}(\varphi(good))(x)$ yields the answer $\{a, e\}$. Consider now the alternating fixpoint sequence in the computation of P^{wf} on the same input (only the positive facts of *bad* and *good* are listed, because G does not change and $answer = good$).

	bad	*good*
$\bar{\mathbf{I}}_0$	$\emptyset$	$\emptyset$
$\bar{\mathbf{I}}_1$	$\{b, c, d, e\}$	$\{a, b, c, d, e\}$
$\bar{\mathbf{I}}_2$	$\emptyset$	$\{a\}$
$\bar{\mathbf{I}}_3$	$\{b, c, d\}$	$\{a, b, c, d, e\}$
$\bar{\mathbf{I}}_4$	$\emptyset$	$\{a, e\}$
$\bar{\mathbf{I}}_5$	$\{b, c, d\}$	$\{a, b, c, d, e\}$
$\bar{\mathbf{I}}_6$	$\emptyset$	$\{a, e\}$

Thus

$$\varphi(\emptyset) = (\bar{\mathbf{I}}_2)^1(good),$$
$$\varphi^2(\emptyset) = (\bar{\mathbf{I}}_4)^1(good)$$

and

$$(\bar{\mathbf{I}}_4)^1(answer) = \mu_{good}(\varphi(good))(x).$$

The relative expressive power of the various languages discussed in this chapter is summarized in Fig. 15.3. The arrows indicate strict inclusion. For a view of these languages in a larger context, see also Figs. 18.4 and 18.5 at the end of Part E.

The Impact of Order

Finally we look at the impact of order on the expressive power of the various datalog$^\neg$ semantics. As we will discuss at length in Chapter 17, the assumption that databases are ordered can have a dramatic impact on the expressive power of languages like *fixpoint* or *while*. The datalog$^\neg$ languages are no exception. The effect of order is spectacular. With this assumption, it turns out that semipositive datalog$^\neg$ is (almost) as powerful as stratified datalog$^\neg$ and datalog$^\neg$ with well-founded semantics. The "almost" comes from a

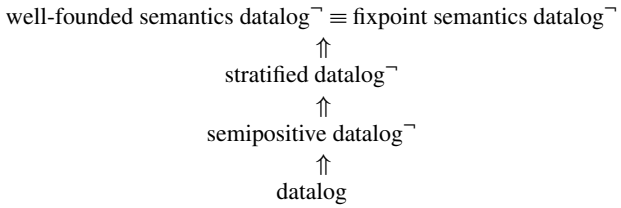


Figure 15.3: Relative expressive power of datalog$^{(\neg)}$ languages

technicality concerning the order: We also need to assume that the minimum and maximum constants are explicitly given. Surprisingly, these constants, which can be computed with a first order query if *succ* is given, cannot be computed with semipositive programs (see Exercise 15.29).

The next lemma states that semipositive programs express the *fixpoint* queries on ordered databases with *min* and *max* (i.e., databases with a predicate *succ* providing a successor relation among all constants, and unary relations *min* and *max* containing the smallest and the largest constant).

Lemma 15.4.7 The semipositive datalog$^\neg$ programs express precisely the *fixpoint* queries on ordered databases with *min* and *max*.

Crux Let q be a *fixpoint* query over database schema $\mathbf{R}$. Because q is a *fixpoint* query, there is a first-order formula $\varphi(T)$, positive in T, such that q is defined by $\mu_T(\varphi(T))(u)$, where u is a vector of variables and constants. Because T is positive in $\varphi(T)$, we can assume that $\varphi(T)$ is in prenex normal form $Q_1x_1Q_2x_2\ldots Q_kx_k(\psi)$, where ψ is a quantifier free formula in disjunctive normal form and T is not negated in ψ. We show by induction on k that there exists a semipositive datalog$^\neg$ program P_φ with an *idb* relation $answer_\varphi$ defining $\mu_T(\varphi(T))$ [the last selection and projection needed to obtain the final answer $\mu_T(\varphi(T))(u)$ pose no problem]. Suppose $k = 0$ (i.e., $\varphi = \psi$). Then P_φ is the nr-datalog$^\neg$ program corresponding to ψ, where the answer relation is T. Because ψ is quantifier free and T is not negated in ψ, P_φ is clearly semipositive. Next suppose the statement is true for some $k \geq 0$, and let $\varphi(T)$ have quantifier depth $k+1$. There are two cases:

(i) $\varphi = \exists x\psi(x, v)$, where ψ has quantifier depth k. Then P_φ contains the rules of P_ψ, where T is replaced in heads of rules by a new predicate T' and one additional rule

$$T(v) \leftarrow T'(x, v).$$

(ii) $\varphi = \forall x\psi(x, v)$, where ψ has quantifier depth k. Then P_φ consists, again, of P_ψ, where T is replaced in heads of rules by a new predicate T', with the following rules added:

$$\begin{aligned} R'(x, v) &\leftarrow T'(x, v), min(x) \\ R'(x', v) &\leftarrow R'(x, v), succ(x, x'), T'(x', v) \\ T(v) &\leftarrow R'(x, v), max(x), \end{aligned}$$

where R' is a new auxiliary predicate. Thus the program steps through all x's using the successor relation *succ*, starting from the minimum constant. If the maximum constant is reached, then $T'(x, v)$ is satisfied for all x, and $T(v)$ is inferred.

This completes the induction. ■

As we shall see in Chapter 17, *fixpoint* expresses on ordered databases exactly the

queries computable in time polynomial in the size of the database (i.e., QPTIME). Thus we obtain the following result. In comparing well-founded semantics with the others, we take the positive portion of the well-founded semantics as the answer.

THEOREM 15.4.8 Stratified datalog$^\neg$ and datalog$^\neg$ with well-founded semantics are equivalent on ordered databases and express exactly QPTIME. They are also equivalent to semipositive datalog$^\neg$ on ordered databases with *min* and *max* and express exactly QPTIME.

15.5 Negation as Failure in Brief

In our presentation of datalog in Chapter 12, we saw that the minimal model and least fixpoint semantics have an elegant proof-theoretic counterpart based on SLD resolution. One might naturally wonder if such a counterpart exists in the case of datalog$^\neg$. The answer is yes and no. Such a proof-theoretic approach has indeed been proposed and is called negation as failure. This was originally developed for logic programming and predates stratified and well-founded semantics. Unfortunately, the approach has two major drawbacks. The first is that it results in a proof-building procedure that does not always terminate. The second is that it is not the exact counterpart of any other existing semantics. The semantics that has been proposed as a possible match is "Clark's completion," but the match is not perfect and Clark's completion has its own problems. We provide here only a brief and informal presentation of negation as failure and the related Clark's completion.

The idea behind negation as failure is simple. We would like to infer a negative fact $\neg A$ if A cannot be proven by SLD resolution. Thus $\neg A$ would then be proven by the failure to prove A. Unfortunately, this is generally noneffective because SLD derivations may be arbitrarily long, and so one cannot check in finite time[2] that there is no proof of A by SLD resolution. Instead we have to use a weaker notion of negation by failure, which can be checked. This is done as follows. A fact $\neg A$ is proven if all SLD derivations starting from the goal $\leftarrow A$ are finite and none produces an SLD refutation for $\leftarrow A$. In other words, A *finitely fails*. This procedure applies to ground atoms A only. It gives rise to a proof procedure called SLDNF resolution. Briefly, SLDNF resolution extends SLD resolution as follows. Refutations of positive facts proceed as for SLD resolution. Whenever a negative ground goal $\leftarrow \neg A$ has to be proven, SLD resolution is applied to $\leftarrow A$, and $\neg A$ is proven if the SLD resolution finitely fails for $\leftarrow A$. The idea of SLDNF seems appealing as the proof-theoretic version of the closed world assumption. However, as illustrated next, it quickly leads to significant problems.

EXAMPLE 15.5.1 Consider the usual program P_{TC} for transitive closure of a graph:

$$T(x, y) \leftarrow G(x, y)$$
$$T(x, y) \leftarrow G(x, z), T(z, y)$$

---

[2] Because databases are finite, one can develop mechanisms to bound the expansion. We ignore this aspect here.

Consider the instance **I** where G has edges $\{\langle a, b\rangle, \langle b, a\rangle, \langle c, a\rangle\}$. Clearly, $\{\langle a, c\rangle\}$ is not in the transitive closure of G, and so not in T, by the usual datalog semantics. Suppose we wish to prove the fact $\neg T(a, c)$, using negation as failure. We have to show that SLD resolution finitely fails on $T(a, c)$, with the preceding program and input. Unfortunately, SLD resolution can enter a negative loop when applied to $\leftarrow T(a, c)$. One obtains the following SLD derivation:

1. $\leftarrow T(a, c)$;
2. $\leftarrow G(a, z), T(z, c)$, using the second rule;
3. $\leftarrow T(b, c)$, using the fact $G(a, b)$;
4. $\leftarrow G(b, z), T(z, c)$ using the second rule;
5. $\leftarrow T(a, c)$ using the fact $G(b, a)$.

Note that the last goal is the same as the first, so this can be extended to an infinite derivation. It follows that SLD resolution does not finitely fail on $\leftarrow T(a, c)$, so SLDNF does not yield a proof of $\neg T(a, c)$. Moreover, it has been shown that this does not depend on the particular program used to define transitive closure. In other words, there is *no* datalog$^{\neg}$ program that under SLDNF can prove the positive and negative facts true of the transitive closure of a graph.

The preceding example shows that SLDNF can behave counterintuitively, even in some simple cases. The behavior is also incompatible with all the semantics for negation that we have discussed so far. Thus one cannot hope for a match between SLDNF and these semantics.

Instead a semantics called Clark's completion has been proposed as a candidate match for negation as failure. It works as follows. For a datalog$^{\neg}$ program P, the *completion of* P, *comp(P)*, is constructed as follows. For each *idb* predicate R, each rule

$$\rho : R(u) \leftarrow L_1(v_1), \ldots, L_n(v_n)$$

defining R is rewritten so there is a uniform set of distinct variables in the rule head and so all free variables in the body are existentially quantified:

$$\rho' : R(u') \leftarrow \exists v'(x_1 = t_1 \wedge \cdots \wedge x_k = t_k \wedge L_1(v_1) \wedge \cdots \wedge L_n(v_n)).$$

(If the head of ρ has distinct variables for all coordinates, then the equality atoms can be avoided. If repeated variables or constants occur, then equality must be used.) Next, if the rewritten rules for R are $\rho'_1, \ldots, \rho'_l$, the *completion* of R is formed by

$$\forall u'(R(u') \leftrightarrow body(\rho'_1) \vee \cdots \vee body(\rho'_l)).$$

Intuitively, this states that ground atom $R(w)$ is true iff it is supported by one of the rules defining R. Finally the completion of P is the set of completions of all *idb* predicates of P, along with the axioms of equality, if needed.

The semantics of P is now defined by the following: A is true iff it is a logical consequence of $comp(P)$. A first problem now is that $comp(P)$ is not always consistent; in fact, its consistency is undecidable. What is the connection between SLDNF and Clark's completion? Because SLDNF is consistent (it clearly cannot prove A and $\neg A$) and $comp(P)$ is not so always, SLDNF is not always complete with respect to $comp(P)$. For consistent $comp(P)$, it can be shown that SLDNF resolution is sound. However, additional conditions must be imposed on the datalog$^\neg$ programs for SLDNF resolution to be complete.

Consider again the transitive closure program P_{TC} and input instance $\mathbf{I}$ of Example 15.5.1. Then the completion of T is equivalent to

$$T(x, y) \leftrightarrow G(x, y) \vee \exists z(G(x, z) \wedge T(z, y)).$$

Note that neither $T(a, c)$ nor $\neg T(a, c)$ are consequences of $comp(P_{TC,\mathbf{I}})$.

In summary, negation as failure does not appear to provide a convincing proof-theoretic counterpart to the semantics we have considered. The search for more successful proof-theoretic approaches is an active research area. Other proposals are described briefly in the Bibliographic Notes.

Bibliographic Notes

The notion of a stratified program is extremely natural. Not surprisingly, it was proposed independently by quite a few investigators [CH85, ABW88, Lif88, VanG86]. The independence of the semantics from a particular stratification (Theorem 15.2.10) was shown in [ABW88].

Research on well-founded semantics, and the related notion of a 3-stable model, has its roots in investigations of stable and default model semantics. Although formulated somewhat differently, the notion of a stable/default model is equivalent to that of a total 3-stable model [Prz90]. Stable model semantics was introduced in [GL88], and default model semantics was introduced in [BF87, BF88]. Stable semantics is based on Moore's autoepistemic logic [Moo85], and default semantics is based on Reiter's default logic [Rei80]. The equivalence between autoepistemic and default logic in the general case has been shown in [Kon88]. The equivalence between stable model semantics and default model semantics was shown in [BF88].

Several equivalent definitions of the well-founded semantics have been proposed. The definition used in this chapter comes from [Prz90]. The alternating fixpoint computation we described is essentially the same as in [VanG89]. Alternative procedures for computing the well-founded semantics are exhibited in [BF88, Prz89]. Historically, the first definition of well-founded semantics was proposed in [VanGRS88, VanGRS91]. This is described in Exercise 15.24.

The fact that well-founded and stratified semantics agree on stratifiable datalog$^\neg$ programs (Theorem 15.3.11) was shown in [VanGRS88].

Both the stratified and well-founded semantics were originally introduced for general logic programming, as well as the more restricted case of datalog. In the context of logic programming, both semantics have expressive power equivalent to the arithmetic hierarchy [AW88] and are thus noneffective.

The result that datalog$^\neg$ with well-founded semantics expresses exactly the *fixpoint*

queries is shown in [VanG89]. Citation [FKL87] proves that for every datalog¬ program P there is a *total* datalog¬ program Q such that the positive portions of $P^{wf}(\mathbf{I})$ and $Q^{wf}(\mathbf{I})$ coincide for every $\mathbf{I}$. The fact that stratified datalog¬ is weaker than *fixpoint*, and therefore weaker than well-founded semantics, was shown in [Kol91], making use of earlier results from [Dal87] and [CH82]. In particular, Lemma 15.4.1 is based on Lemma 3.9 in [CH82]. The result that semipositive datalog¬ expresses QPTIME on ordered databases with *min* and *max* is due to [Pap85].

The investigation of negation as failure was initiated in [Cla78], in connection with general logic programming. In particular, SLDNF resolution as well as Clark's completion are introduced there. The fact that there is no datalog¬ program for which the positive and negative facts about the transitive closure of the graph can be proven by SLDNF resolution was shown in [Kun88]. Other work related to Clark's completion can be found in [She88, Llo87, Fit85, Kun87].

Several variations of SLDNF resolutions have been proposed. SLS resolution is introduced in [Prz88] to deal with stratified programs. An exact match is achieved between stratified semantics and the proof procedure provided by SLS resolution. Although SLS resolution is effective in the context of (finite) databases, it is not so when applied to general logic programs, with function symbols. To deal with this shortcoming, several restrictions of SLS resolution have been proposed that are effective in the general framework [KT88, SI88].

Several proof-theoretic approaches corresponding to the well-founded semantics have been proposed. SLS resolution is extended from stratified to arbitrary datalog¬ programs in [Prz88], under well-founded semantics. Independently, another extension of SLS resolution called global SLS resolution is proposed in [Ros89], with similar results. These proposals yield noneffective resolution procedures. An effective procedure is described in [BL90].

In [SZ90], an interesting connection between nondeterminism and stable models of a program (i.e., total 3-stable models; see also Exercise 15.20) is pointed out. Essentially, it is shown that the stable models of a datalog¬ program can be viewed as the result of a natural nondeterministic choice. This uses the choice construct introduced earlier in [KN88]. Another use of nondeterminism is exhibited in [PY92], where an extension of well-founded semantics is provided, which involves the nondeterministic choice of a fixpoint of a datalog¬ program. This is called *tie-breaking* semantics. A discussion of nondeterminism in deductive databases is provided in [GPSZ91].

Another semantics in the spirit of well-founded is the valid model semantics introduced in [BRSS92]. It is less conservative than well-founded semantics, in the sense that all facts that are positive in well-founded semantics are also positive in the valid model semantics, but the latter generally yields more positive facts than well-founded semantics.

There are a few prototypes (but no commercial system) implementing stratified datalog¬. The language LDL [NT89, BNR+87, NK88] implements, besides the stratified semantics for datalog¬, an extension to complex objects (see also Chapter 20). The implementation uses heuristics based on the magic set technique described in Chapter 13. The language NAIL! (Not Yet Another Implementation of Logic!), developed at Stanford, is another implementation of the stratified semantics, allowing function symbols and a set construct. The implementation of NAIL! [MUG86, Mor88] uses a battery of evaluation techniques, including magic sets. The language EKS [VBKL89], developed at

ECRC (European Computer-Industry Research Center) in Munich, implements the stratified semantics and extensions allowing quantifiers in rule bodies, aggregate functions, and constraint specification. The CORAL system [RSS92, RSSS93] provides a database programming language that supports both imperative and deductive capabilities, including stratification. An implementation of well-founded semantics is described in [CW92].

Nicole Bidoit's survey on negation in databases [Bid91b], as well as her book on datalog [Bid91a], provided an invaluable source of information and inspired our presentation of the topic.

Exercises

Exercise 15.1

(a) Show that, for datalog$^\neg$ programs P, the immediate consequence operator T_P is not always monotonic.

(b) Exhibit a datalog$^\neg$ program P (using negation at least once) such that T_P *is* monotonic.

(c) Show that it is decidable, given a datalog$^\neg$ program P, whether T_P is monotonic.

Exercise 15.2 Consider the datalog$^\neg$ program $P_3 = \{p \leftarrow \neg r;\ r \leftarrow \neg p;\ p \leftarrow \neg p, r\}$. Verify that T_{P_3} has a least fixpoint, but T_{P_3} does not converge when starting on $\emptyset$.

Exercise 15.3

(a) Exhibit a datalog$^\neg$ program P and an instance $\mathbf{K}$ over $sch(P)$ such that $\mathbf{K}$ is a model of Σ_P but not a fixpoint of T_P.

(b) Show that, for datalog$^\neg$ programs P, a minimal fixpoint of T_P is not necessarily a minimal model of Σ_P and, conversely, a minimal model of Σ_P is not necessarily a minimal fixpoint of T_P.

Exercise 15.4 Prove Lemma 15.2.8.

Exercise 15.5 Consider a database for the Parisian metro and bus lines, consisting of two relations *Metro*[*Station, Next-Station*] and *Bus*[*Station, Next-Station*]. Write stratifiable datalog$^\neg$ programs to answer the following queries.

(a) Find the pairs of stations $\langle a, b\rangle$ such that one can go from a to b by metro but not by bus.

(b) A *pure bus path* from a to b is a bus itinerary from a to b such that for all consecutive stops c, d along the way, one cannot go from c to d by metro. Find the pairs of stations $\langle a, b\rangle$ such that there is a pure bus path from a to b.

(c) Find the pairs of stations $\langle a, b\rangle$ such that b can be reached from a by some combination of metro or bus, but not by metro or bus alone.

(d) Find the pairs of stations $\langle a, b\rangle$ such that b can be reached from a by some combination of metro or bus, but there is no pure bus path from a to b.

(e) The metro is useless in a bus path from a to b if by taking the metro at any intermediate point c one can return to c but not reach any other station along the path. Find the pairs of stations $\langle a, b\rangle$ such that the metro is useless in all bus paths connecting a and b.

Exercise 15.6 The semantics of stratifiable datalog¬ programs can be extended to infinite databases as follows. Let P be a stratifiable datalog¬ program and let $\sigma = P^1 \dots P^n$ be a stratification for P. For each (finite or infinite) instance $\mathbf{I}$ over $edb(P)$, $\sigma(\mathbf{I})$ is defined similarly to the finite case. More precisely, consider the sequence

$$\mathbf{I}_0 = \mathbf{I}$$
$$\mathbf{I}_i = P^i(\mathbf{I}_{i-1}|edb(P^i))$$

where

$$P^i(\mathbf{I}_{i-1}|edb(P^i)) = \cup_{j>0} T^j_{P_i}(\mathbf{I}_{i-1}|edb(P^i)).$$

Note that the definition is now noneffective because $P^i(\mathbf{I}_{i-1}|edb(P^i))$ may be infinite.

Consider a database consisting of one binary relation *succ* providing a successor relation on an infinite set of constants. Clearly, one can identify these constants with the positive integers.

(a) Write a stratifiable datalog¬ program defining a unary relation *prime* containing all constants in *succ* corresponding to primes.

(b) Write a stratifiable datalog¬ program P defining a 0-ary relation *Fermat*, which is true iff Fermat's Last Theorem[3] is true. (No shortcuts, please: The computation of the program should provide a proof of Fermat's Last Theorem, not just coincidence of truth value!)

Exercise 15.7 Prove Theorem 15.2.2.

Exercise 15.8 A datalog¬ program is *nonrecursive* if its precedence graph is acyclic. Show that every nonrecursive stratifiable datalog¬ program is equivalent to an nr-datalog¬ program, and conversely.

Exercise 15.9 Let $(A, <)$ be a partially ordered set. A listing $a_1, \dots, a_n$ of the elements in A is *compatible with* $<$ iff for $i < j$ it is not the case that $a_j < a_i$. Let σ', σ'' be listings of A compatible with $<$. Prove that one can obtain σ'' from σ' by a sequence of exchanges of adjacent elements a_l, a_m such that $a_l \not< a_m$ and $a_m \not< a_l$.

Exercise 15.10 Prove Lemma 15.2.9.

Exercise 15.11 (Supported models) Prove that there exist stratified datalog¬ programs P_1, P_2 such that $sch(P_1) = sch(P_2)$, $\Sigma_{P_1} \equiv \Sigma_{P_2}$, and there is a minimal model $\mathbf{I}$ of Σ_{P_1} such that $\mathbf{I}$ is a supported model for P_1, but not for P_2. (In other words, the notion of supported model depends not only on Σ_P, but also on the syntax of P.)

Exercise 15.12 Prove part (b) of Proposition 15.2.11.

Exercise 15.13 Prove Proposition 15.2.12.

♠ **Exercise 15.14** [Bid91b] (Local stratification) The following extension of the notion of stratification has been proposed for general logic programs [Prz86]. This exercise shows that local stratification is essentially the same as stratification for the datalog¬ programs considered in this chapter (i.e., without function symbols).

---

[3] Fermat's Last Theorem: There is no $n > 2$ such that the equation $a^n + b^n = c^n$ has a solution in the positive integers.

A datalog$^\neg$ program P is *locally stratified* iff for each $\mathbf{I}$ over $edb(P)$, $ground(P_\mathbf{I})$ is stratified. [An example of a locally stratified logic program with function symbols is $\{even(0) \leftarrow; even(s(x)) \leftarrow \neg even(x)\}$.] The semantics of a locally stratified program P on input $\mathbf{I}$ is the semantics of the stratified program $ground(P_\mathbf{I})$.

(a) Show that, if the rules of P contain no constants, then P is locally stratified iff it is stratified.

(b) Give an example of a datalog$^\neg$ program (with constants) that is locally stratified but not stratified.

(c) Prove that, for each locally stratified datalog$^\neg$ program P, there exists a stratified datalog$^\neg$ program equivalent to P.

Exercise 15.15 Let α and β be propositional Boolean formulas (using $\wedge, \vee, \neg, \rightarrow$). Prove the following:

(a) If α and β are equivalent with respect to 3-valued instances, then they are equivalent with respect to 2-valued instances.

(b) If α and β are equivalent with respect to 2-valued instances, they are not necessarily equivalent with respect to 3-valued instances.

Exercise 15.16 Prove Lemma 15.3.4.

Exercise 15.17 Let P be a datalog$^\neg$ program. Recall the definition of positivized ground version of P given $\mathbf{I}$, denoted $pg(P, \mathbf{I})$, where $\mathbf{I}$ is a 3-valued instance. Prove the following:

(a) If $\mathbf{I}$ is total, then $pg(P, \mathbf{I})$ is total.

(b) Let $\{\mathbf{I}_i\}_{i \geq 0}$ be the sequence of instances defined by

$$\begin{aligned} \mathbf{I}_0 &= \bot \\ \mathbf{I}_{i+1} &= pg(P, \mathbf{I}_i)(\bot) = conseq_P(\mathbf{I}_i). \end{aligned}$$

Prove that

$$\mathbf{I}_0 \prec \mathbf{I}_2 \cdots \prec \mathbf{I}_{2i} \prec \mathbf{I}_{2i+2} \prec \cdots \prec \mathbf{I}_{2i+1} \prec \mathbf{I}_{2i-1} \prec \cdots \prec \mathbf{I}_1.$$

Exercise 15.18 Exhibit a datalog$^\neg$ program that yields the complement of the transitive closure under well-founded semantics.

Exercise 15.19 Prove that for each datalog$^\neg$ program P and instance $\mathbf{I}$ over $edb(P)$, $P^{wf}(\mathbf{I})$ is a minimal 3-valued model of P whose restriction to $edb(P)$ equals $\mathbf{I}$.

♠ **Exercise 15.20** A total 3-stable model of a datalog$^\neg$ program P is called a *stable model of* P [GL88] (also called a *default model* [BF87, BF88]).

(a) Provide examples of datalog$^\neg$ programs that have (1) no stable models, (2) a unique stable model, and (3) several stable models.

(b) Show that P^{wf} is total iff all 3-stable models are total.

(c) Prove that, if P^{wf} is total, then P has a unique stable model, but the converse is false.

♠ **Exercise 15.21** [BF88] Let P be a datalog$^\neg$ program and $\mathbf{I}$ an instance over $edb(P)$. Prove that the problem of determining whether $P_\mathbf{I}$ has a stable model is NP-complete in the size of $P_\mathbf{I}$.

Exercise 15.22 Give an example of a datalog$^\neg$ program P such that P is not stratified but P^{wf} is total.

★ **Exercise 15.23** Prove that it is undecidable if the well-founded semantics of a given datalog$^\neg$ program P is always total. That is, it is undecidable whether, for each instance $\mathbf{I}$ over $edb(P)$, $P_{\mathbf{I}}^{wf}$ is total.

♠ **Exercise 15.24** [VanGRS88] This exercise provides an alternative (and historically first) definition of well-founded semantics. Let L be a ground literal. The *complement of* L is $\neg A$ if $L = A$ and A if $L = \neg A$. If $\mathbf{I}$ is a set of ground literals, we denote by $\neg.\mathbf{I}$ the set of complements of the literals in $\mathbf{I}$. A set $\mathbf{I}$ of ground literals is *consistent* iff $\mathbf{I} \cap \neg.\mathbf{I} = \emptyset$. Let P be a datalog$^\neg$ program. The immediate consequence operator T_P of P is extended to operate on sets of (positive and negative) ground literals as follows. Let $\mathbf{I}$ be a set of ground literals. $T_P(\mathbf{I})$ consists of all literals A for which there is a ground rule of P, $A \leftarrow L_1, \ldots, L_k$, such that $L_i \in \mathbf{I}$ for each i. Note that T_P can produce an inconsistent set of literals, which therefore does not correspond to a 3-valued model. Now let $\mathbf{I}$ be a set of ground literals and $\mathbf{J}$ a set of positive ground literals. $\mathbf{J}$ is said to be *an unfounded set* of P with respect to $\mathbf{I}$ if for each $A \in \mathbf{J}$ and ground rule r of P with A in the head, at least one of the following holds:

- the complement of some literal in the body of r is in $\mathbf{I}$; or
- some positive literal in the body of r is in $\mathbf{J}$.

Intuitively, this means that if all atoms of $\mathbf{I}$ are assumed true and all atoms in $\mathbf{J}$ are assumed false, then no atom of $\mathbf{J}$ is true under one application of T_P.

Let the *greatest unfounded set of P with respect to* $\mathbf{I}$ be the union of all unfounded sets of P with respect to $\mathbf{I}$, denoted $U_P(\mathbf{I})$. Next consider the operator W_P on sets of ground literals defined by

$$W_P(\mathbf{I}) = T_P(\mathbf{I}) \cup \neg.U_P(\mathbf{I}).$$

Prove the following:

(a) The greatest unfounded set $U_P(\mathbf{I})$ of P with respect to $\mathbf{I}$ is an unfounded set.

(b) The operator W_P is monotonic (with respect to set inclusion).

(c) The least fixpoint of W_P is consistent.

(d) The least fixpoint of W_P equals P^{wf}.

♠ **Exercise 15.25** [VanG89] Let P be a datalog$^\neg$ program. If $\mathbf{I}$ is a set of ground literals, let $P(\mathbf{I}) = T_P^\omega(\mathbf{I})$, where T_P is the immediate consequence operator on sets of ground literals defined in Exercise 15.24. Furthermore, $\overline{P}(\mathbf{I})$ denotes the complement of $P(\mathbf{I})$ [i.e., $\mathbf{B}(P, \mathbf{I}) - P(\mathbf{I})$]. Consider the sequence of sets of negative facts defined by

$$\begin{aligned} \mathbf{N}_0 &= \emptyset, \\ \mathbf{N}_{i+1} &= \neg.\overline{P}(\neg.\overline{P}(\mathbf{N}_i)). \end{aligned}$$

The intuition behind the definition is the following. $\mathbf{N}_0$ is an underestimate of the set of negative facts in the well-founded model. Then $P(\mathbf{N})$ is an underestimate of the positive facts, and the negated complement $\neg.\overline{P}(\mathbf{N})$ is an overestimate of the negative facts. Using this overestimate, one can infer an overestimate of the positive facts, $P(\neg.\overline{P}(\mathbf{N}))$. Therefore $\neg.\overline{P}(\neg.\overline{P}(\mathbf{N}))$ is now a new underestimate of the negative facts containing the previous underestimate. So $\{\mathbf{N}_i\}_{i \geq 0}$ is

an increasing sequence of underestimates of the negative facts, which converges to the negative facts in the well-founded model. Formally prove the following:

(a) The sequence $\{\mathbf{N}_i\}_{i\geq 0}$ is increasing.

(b) Let $\mathbf{N}$ be the limit of the sequence $\{\mathbf{N}_i\}_{i\geq 0}$ and $\mathbf{K} = \mathbf{N} \cup P(\mathbf{N})$. Then $\mathbf{K} = P^{wf}$.

(c) Explain the connection between the sequence $\{\mathbf{N}_i\}_{i\geq 0}$ and the sets of negative facts in the sequence $\{\mathbf{I}_i\}_{i\geq 0}$ defined in the alternating fixpoint computation of P^{wf} in the text.

(d) Suppose the definition of the sequence $\{\mathbf{N}_i\}_{i\geq 0}$ is modified such that $\mathbf{N}_0 = \neg.\mathbf{B}(P)$ (i.e., all facts are negative at the start). Show that for each $i \geq 0$, $\mathbf{N}_i = \neg.(\mathbf{I}_{2i})^0$.

Exercise 15.26 Let P be a datalog$^\neg$ program. Let T_P be the immediate consequence operator on sets of ground literals, defined in Exercise 15.24, and let $\bar{T}_P$ be defined by $\bar{T}_P(\mathbf{I}) = \mathbf{I} \cup T_P(\mathbf{I})$. Given a set $\mathbf{I}$ of ground literals, let $P(\mathbf{I})$ denote the limit of the increasing sequence $\{\bar{T}_P^i(\mathbf{I})\}_{i>0}$. A set $\mathbf{I}^-$ of negative ground literals is *consistent with respect to* P if $P(\mathbf{I}^-)$ is consistent. $\mathbf{I}^-$ is *maximally consistent with respect to* P if it is maximal among the sets of negative literals consistent with P. Investigate the connection between maximal consistency, 3-stable models, and well-founded semantics:

(a) Is $\neg.\mathbf{I}^0$ maximally consistent for every 3-stable model $\mathbf{I}$ of P?

(b) Is $P(\mathbf{I}^-)$ a 3-stable model of P for every $\mathbf{I}^-$ that is maximally consistent with respect to P?

(c) Is $\neg.(P^{wf})^0$ the intersection of all sets $\mathbf{I}^-$ that are maximally consistent with respect to P?

Exercise 15.27 Refer to the proof of Lemma 15.4.4.

(a) Outline a proof that $conseq_P$ can be simulated by a *while*$^+$ program.

(b) Provide a full description of the timestamping technique outlined in the proof of Lemma 15.4.4.

Exercise 15.28 Show that every query definable by stratified datalog$^\neg$ is a *fixpoint* query.

Exercise 15.29 Consider an ordered database (i.e., with binary relation *succ* providing a successor relation on the constants). Prove that the minimum and maximum constants cannot be computed using a semipositive program.

★ **Exercise 15.30** Consider the game trees and *winning* query described in Section 15.4.

(a) Show that *winning* is true on the game trees $G_{2i,k}$ and false on the game trees $G'_{2i,k}$, for $i > 0$.

(b) Prove that the *winning* query on game trees is defined by the *fixpoint* query exhibited in Section 15.4.