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:

$$\begin{aligned}
St_Reachable(Odeon, Louvre) \leftarrow\ & St_Reachable(Odeon, Chatelet), \\
& Links(1, Chatelet, Louvre)
\end{aligned}$$

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. ■

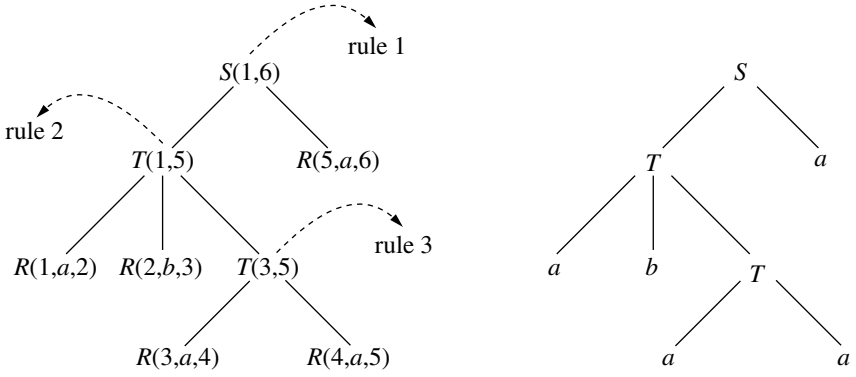
12.4 Proof-Theoretic Approach

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:

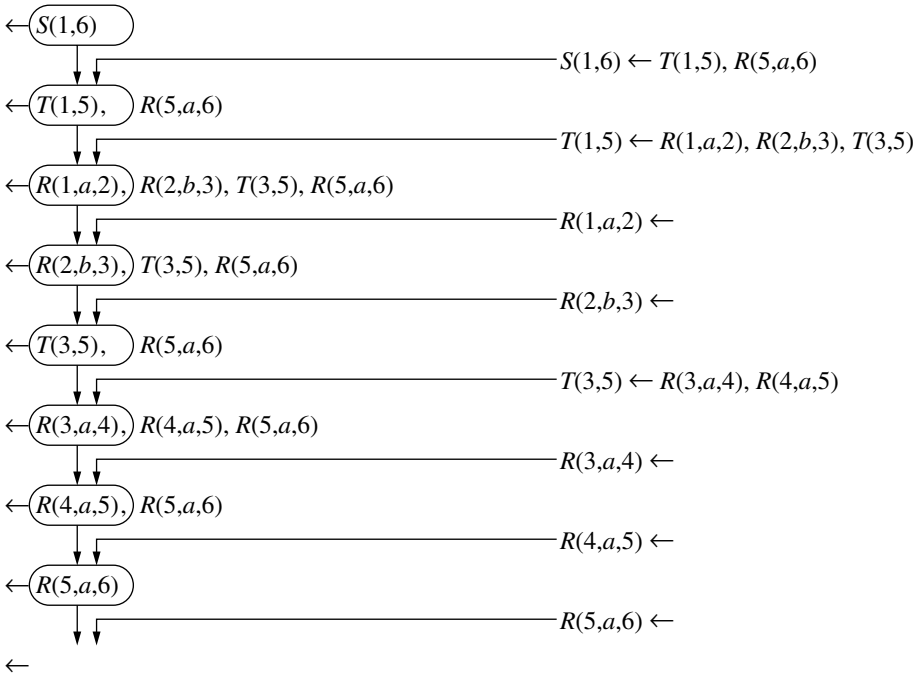


Figure 12.4: SLD ground refutation

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

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

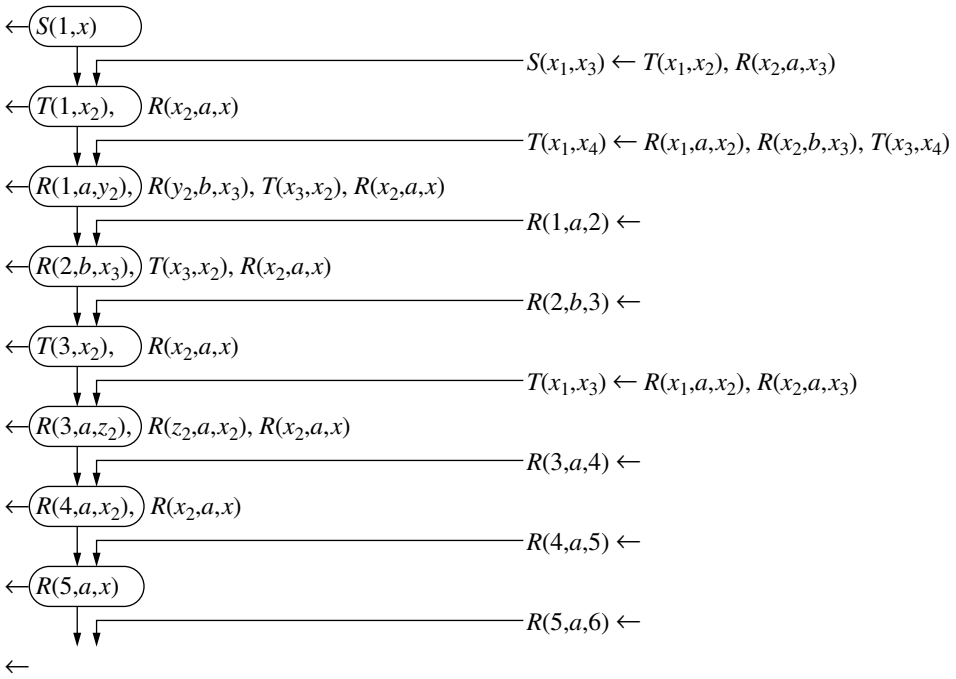


Figure 12.5: SLD refutation

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. ■

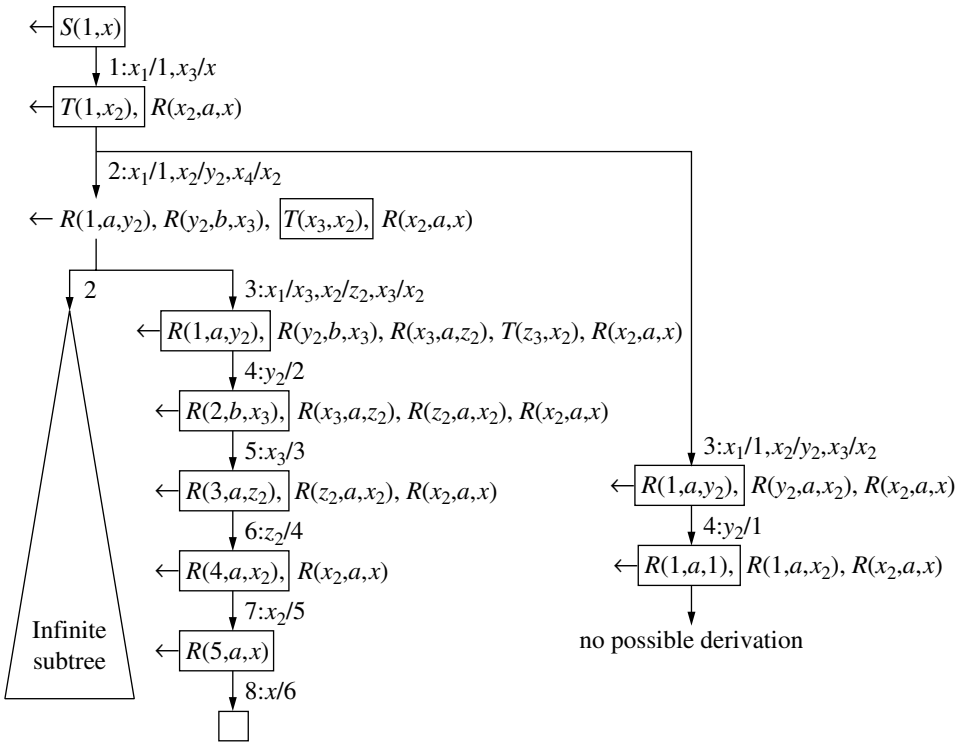


Figure 12.6: SLD tree

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.