15 Negation in Datalog

Alice: *I thought we already talked about negation.*
Sergio: *Yes, but they say you don't think by fixpoint.*
Alice: *Humbug, I just got used to it!*
Riccardo: *So we have to tell you how you* really *think.*
Vittorio: *And convince you that our explanation is well founded!*

As originally introduced in Chapter 12, datalog is a toy language that expresses many interesting recursive queries but has serious shortcomings concerning expressive power. Because it is monotonic, it cannot express simple relational algebra queries such as the difference of two relations. In the previous chapter, we considered one approach for adding negation to datalog that led to two procedural languages—namely, inflationary datalog$^\neg$ and datalog$^{\neg\neg}$. In this chapter, we take a different point of view inspired by nonmonotonic reasoning that attempts to view the semantics of such programs in terms of a natural reasoning process.

This chapter begins with illustrations of how the various semantics for datalog do not naturally extend to datalog$^\neg$. Two semantics for datalog$^\neg$ are then considered. The first, called *stratified*, involves a syntactic restriction on programs but provides a semantics that is natural and relatively easy to understand. The second, called *well founded*, requires no syntactic restriction on programs, but the meaning associated with some programs is expressed using a 3-valued logic. (In this logic, facts are true, false, or unknown.) With respect to expressive power, well-founded semantics is equivalent to the *fixpoint* queries, whereas the stratified semantics is strictly weaker. A proof-theoretic semantics for datalog$^\neg$, based on *negation as failure*, is discussed briefly at the end of this chapter.

15.1 The Basic Problem

Suppose that we want to compute the pairs of disconnected nodes in a graph G (i.e., we are interested in the *complement* of the transitive closure of a graph whose edges are given by a binary relation G). We already know how to define the transitive closure of G in a relation T using the datalog program P_{TC} of Chapter 12:

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

To define the complement CT of T, we are naturally tempted to use negation as we

did in Chapter 5. Let P_{TCcomp} be the result of adding the following rule to P_{TC}:

$$CT(x, y) \leftarrow \neg T(x, y).$$

To simplify the discussion, we generally assume an active domain interpretation of datalog$^\neg$ rules.

In this example, negation appears to be an appealing addition to the datalog syntax. The language datalog$^\neg$ is defined by allowing, in bodies of rules, literals of the form $\neg R_i(u_i)$, where R_i is a relation name and u_i is a free tuple. In addition, the equality predicate is allowed, and $\neg = (x, y)$ is denoted by $x \neq y$.

One might hope to extend the model-theoretic, fixpoint, and proof-theoretic semantics of datalog just as smoothly as the syntax. Unfortunately, things are less straightforward when negation is present. We illustrate informally the problems that arise if one tries to extend the least-fixpoint and minimal-model semantics of datalog. We shall discuss the proof-theoretic aspect later.

Fixpoint Semantics: Problems

Recall that, for a datalog program P, the fixpoint semantics of P on input $\mathbf{I}$ is the unique minimal fixpoint of the immediate consequence operator T_P containing $\mathbf{I}$. The immediate consequence operator can be naturally extended to a datalog$^\neg$ program P. For a program P, T_P is defined as follows[1]: For each $\mathbf{K}$ over $sch(P)$, A is $T_P(\mathbf{K})$ if $A \in \mathbf{K}|edb(P)$ or if there exists some instantiation $A \leftarrow A_1, \ldots, A_n$ of a rule in P for which (1) if A_i is a positive literal, then $A_i \in \mathbf{K}$; and (2) if $A_i = \neg B_i$ where B_i is a positive literal, then $B_i \notin \mathbf{K}$. [Note the difference from the immediate consequence operator Γ_P defined for datalog$^\neg$ in Section 14.3: Γ_P is inflationary by definition, (that is, $\mathbf{K} \subseteq \Gamma_P(\mathbf{K})$ for each $\mathbf{K}$ over $sch(P)$, whereas T_P is not.] The following example illustrates several unexpected properties that T_P might have.

EXAMPLE 15.1.1

(a) T_P may not have any fixpoint. For the propositional program $P_1 = \{p \leftarrow \neg p\}$, T_{P_1} has no fixpoint.

(b) T_P may have several minimal fixpoints containing a given input. For example, the propositional program $P_2 = \{p \leftarrow \neg q, q \leftarrow \neg p\}$ has two minimal fixpoints (containing the empty instance): $\{p\}$ and $\{q\}$.

(c) Consider the sequence $\{T_P^i(\emptyset)\}_{i>0}$ for a given datalog$^\neg$ program P. Recall that for datalog, the sequence is increasing and converges to the least fixpoint of T_P. In the case of datalog$^\neg$, the situation is more intricate:

1. The sequence does not generally converge, even if T_P has a least fixpoint. For example, let $P_3 = \{p \leftarrow \neg r; r \leftarrow \neg p; p \leftarrow \neg p, r\}$. Then

---

[1] Given an instance $\mathbf{J}$ over a database schema $\mathbf{R}$ with $\mathbf{S} \subseteq \mathbf{R}$, $\mathbf{J}|\mathbf{S}$ denotes the restriction of $\mathbf{J}$ to $\mathbf{S}$.

T_{P_3} has a least fixpoint $\{p\}$ but $\{T^i_{P_3}(\emptyset)\}_{i>0}$ alternates between $\emptyset$ and $\{p, r\}$ and so does not converge (Exercise 15.2).

2. Even if $\{T^i_P(\emptyset)\}_{i>0}$ converges, its limit is not necessarily a minimal fixpoint of T_P, even if such fixpoints exist. To see this, let $P_4 = \{p \leftarrow p, q \leftarrow q, p \leftarrow \neg p, q \leftarrow \neg p\}$. Now $\{T^i_{P_4}(\emptyset)\}_{i>0}$ converges to $\{p, q\}$ but the least fixpoint of T_{P_4} equals $\{p\}$.

Remark 15.1.2 (Inflationary fixpoint semantics) The program P_4 of the preceding example contains two rules of a rather strange form: $p \leftarrow p$ and $q \leftarrow q$. In some sense, such rules may appear meaningless. Indeed, their logical forms [e.g., $(p \vee \neg p)$] are tautologies. However, rules of the form $R(x_1, \ldots, x_n) \leftarrow R(x_1, \ldots, x_n)$ have a nontrivial impact on the immediate consequence operator T_P. If such rules are added for each *idb* relation R, this results in making T_P inflationary [i.e., $\mathbf{K} \subseteq T_P(\mathbf{K})$ for each $\mathbf{K}$], because each fact is an immediate consequence of itself. It is worth noting that in this case, $\{T^i_P(\mathbf{I})\}_{i>0}$ always converges and the semantics given by its limit coincides with the inflationary fixpoint semantics for datalog$^\neg$ programs exhibited in Chapter 14.

To see the difference between the two semantics, consider again program P_{TCcomp}. The sequence $\{T^i_{P_{TCcomp}}(I)\}_{i>0}$ on input I over G converges to the desired answer (the complement of transitive closure). With the inflationary fixpoint semantics, CT becomes a complete graph at the first iteration (because T is initially empty) and P_{TCcomp} does not compute the complement of transitive closure. Nonetheless, it was shown in Chapter 14 that there is a different (more complicated) datalog$^\neg$ program that computes the complement of transitive closure with the inflationary fixpoint semantics. ■

Model-Theoretic Semantics: Problems

As with datalog, we can associate with a datalog$^\neg$ program P the set Σ_P of CALC sentences corresponding to the rules of P. Note first that, as with datalog, Σ_P always has at least one model containing any given input $\mathbf{I}$. $B(P, \mathbf{I})$ is such a model. [Recall that $B(P, \mathbf{I})$, introduced in Chapter 12, is the instance in which the idb relations contain all tuples with values in $\mathbf{I}$ or P.]

For datalog, the model-theoretic semantics of a program P was given by the unique minimal model of Σ_P containing the input. Unfortunately, this simple solution no longer works for datalog$^\neg$, because uniqueness of a minimal model containing the input is not guaranteed. Program P_2 in Example 15.1.1(b) provides one example of this: $\{p\}$ and $\{q\}$ are distinct minimal models of P_2. As another example, consider the program P_{TCcomp} and an input I for predicate G. Let $\mathbf{J}$ over $sch(P_{TCcomp})$ be such that $\mathbf{J}(G) = I$, $\mathbf{J}(T) \supseteq I$, $\mathbf{J}(T)$ is transitively closed, and $\mathbf{J}(CT) = \{\langle x, y\rangle \mid x, y \text{ occur in } \mathbf{I}, \langle x, y\rangle \notin \mathbf{J}(T)\}$. Clearly, there may be more than one such $\mathbf{J}$, but one can verify that each one is a minimal model of $\Sigma_{P_{TCcomp}}$ satisfying $\mathbf{J}(G) = I$.

It is worth noting the connection between T_P and models of Σ_P: An instance $\mathbf{K}$ over $sch(P)$ is a model of Σ_P iff $T_P(\mathbf{K}) \subseteq \mathbf{K}$. In particular, every fixpoint of T_P is a model of Σ_P. The converse is false (Exercise 15.3).

When for a program P, Σ_P has several minimal models, one must specify which

among them is the model intended to be the solution. To this end, various criteria of "niceness" of models have been proposed that can distinguish the intended model from other candidates. We shall discuss several such criteria as we go along. Unfortunately, none of these criteria suffices to do the job. Moreover, upon reflection it is clear that no criteria can exist that would always permit identification of a unique intended model among several minimal models. This is because, as in the case of program P_2 of Example 15.1.1(b), the minimal models can be completely symmetric; in such cases there is no property that would separate one from the others using just the information in the input or the program.

In summary, the approach we used for datalog, based on equivalent least-fixpoint or minimum-model semantics, breaks down when negation is present. We shall describe several solutions to the problem of giving semantics to datalog$^\neg$ programs. We begin with the simplest case and build up from there.

15.2 Stratified Semantics

This section begins with the restricted case in which negation is applied only to *edb* relations. The semantics for negation is straightforward in this case. We then turn to stratified semantics, which extends this simple case in an extremely natural fashion.

Semipositive Datalog$^\neg$

We consider now *semipositive* datalog$^\neg$ programs, which only apply negation to *edb* relations. For example, the difference of R and R' can be defined by the one-rule program

$$Diff(x) \leftarrow R(x), \neg R'(x).$$

To give semantics to $\neg R'(x)$, we simply use the closed world assumption (see Chapter 2): $\neg R'(x)$ holds iff x is in the active domain and $x \notin R'$. Because R' is an *edb* relation, its content is given by the database and the semantics of the program is clear. We elaborate on this next.

Definition 15.2.1 A datalog$^\neg$ program P is *semipositive* if, whenever a negative literal $\neg R'(x)$ occurs in the body of a rule in P, $R' \in edb(P)$.

As their name suggests, semipositive programs are almost positive. One could eliminate negation from semipositive programs by adding, for each *edb* relation R', a new *edb* relation $\overline{R'}$ holding the complement of R' (with respect to the active domain) and replacing $\neg R'(x)$ by $\overline{R'}(x)$. Thus it is not surprising that semipositive programs behave much like datalog programs. The next result is shown easily and is left for the reader (Exercise 15.7).

Theorem 15.2.2 Let P be a semipositive datalog$^\neg$ program. For every instance $\mathbf{I}$ over $edb(P)$,

(i) Σ_P has a unique minimal model $\mathbf{J}$ satisfying $\mathbf{J}|edb(P) = \mathbf{I}$.

(ii) T_P has a unique minimal fixpoint $\mathbf{J}$ satisfying $\mathbf{J}|edb(P) = \mathbf{I}$.

(iii) The minimum model in (i) and the least fixpoint in (ii) are identical and equal to the limit of the sequence $\{T_P^i(\mathbf{I})\}_{i>0}$.

Remark 15.2.3 Observe that in the theorem, we use the formulation "minimal model satisfying $\mathbf{J}|edb(P) = \mathbf{I}$," whereas in the analogous result for datalog we used "minimal model containing $\mathbf{I}$." Both formulations would be equivalent in the datalog setting because adding tuples to the *edb* predicates would result in larger models because of monotonicity. This is not the case here because negation destroys monotonicity. ■

Given a semipositive datalog$^\neg$ program P and an input $\mathbf{I}$, we denote by $P^{semi-pos}(\mathbf{I})$ the minimum model of Σ_P (or equivalently, the least fixpoint of T_P) whose restriction to $edb(P)$ equals $\mathbf{I}$.

An example of a semipositive program that is neither in datalog nor in CALC is given by

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

This program computes the transitive closure of the complement of G. On the other hand, the foregoing program for the complement of transitive closure is not a semipositive program. However, it can naturally be viewed as the composition of two semipositive programs: the program computing the transitive closure followed by the program computing its complement. Stratification, which is studied next, may be viewed as the closure of semipositive programs under composition. It will allow us to specify, for instance, the composition just described, computing the complement of transitive closure.

Syntactic Restriction for Stratification

We now consider a natural extension of semipositive programs. In semipositive programs, the use of negation is restricted to *edb* relations. Now suppose that we use some *defined* relations, much like views. Once a relation has been defined by some program, other programs can subsequently treat it as an *edb* relation and apply negation to it. This simple idea underlies an important extension to semipositive programs, called stratified programs.

Suppose we have a datalog$^\neg$ program P. Each *idb* relation is defined by one or more rules of P. If we are able to "read" the program so that, for each *idb* relation R', the portion of P defining R' comes before the negation of R' is used, then we can simply compute R' before its negation is used, and we are done. For example, consider program P_{TCcomp} introduced at the beginning of this chapter. Clearly, we intended for T to be defined by the first two rules before its negation is used in the rule defining CT. Thus the first two rules are applied before the third. Such a way of "reading" P is called a stratification of P and is defined next.

Definition 15.2.4 A *stratification* of a datalog$^\neg$ program P is a sequence of datalog$^\neg$ programs $P^1, \ldots, P^n$ such that for some mapping σ from $idb(P)$ to $[1..n]$,

(i) $\{P^1, \ldots, P^n\}$ is a partition of P.

(ii) For each predicate R, all the rules in P defining R are in $P^{\sigma(R)}$ (i.e., in the same program of the partition).

(iii) If $R(u) \leftarrow \ldots R'(v) \ldots$ is a rule in P, and R' is an *idb* relation, then $\sigma(R') \leq \sigma(R)$.

(iv) If $R(u) \leftarrow \ldots \neg R'(v) \ldots$ is a rule in P, and R' is an *idb* relation, then $\sigma(R') < \sigma(R)$.

Given a stratification $P^1, \ldots, P^n$ of P, each P^i is called a *stratum* of the stratification, and σ is called the *stratification mapping*.

Intuitively, a stratification of a program P provides a way of parsing P as a sequence of subprograms $P^1, \ldots, P^n$, each defining one or several *idb* relations. By (iii), if a relation R' is used positively in the definition of R, then R' must be defined earlier or simultaneously with R (this allows recursion!). If the negation of R' is used in the definition of R, then by (iv) the definition of R' must come strictly before that of R.

Unfortunately, not every datalog$^\neg$ program has a stratification. For example, there is no way to "read" program P_2 of Example 15.1.1 so that p is defined before q and q before p. Programs that have a stratification are called *stratifiable*. Thus P_2 is not stratifiable. On the other hand, P_{TCcomp} is clearly stratifiable: The first stratum consists of the first two rules (defining T), and the second stratum consists of the third rule (defining CT using T).

Example 15.2.5 Consider the program P_7 defined by

$$\begin{aligned} r_1 \quad & S(x) \leftarrow R'_1(x), \neg R(x) \\ r_2 \quad & T(x) \leftarrow R'_2(x), \neg R(x) \\ r_3 \quad & U(x) \leftarrow R'_3(x), \neg T(x) \\ r_4 \quad & V(x) \leftarrow R'_4(x), \neg S(x), \neg U(x). \end{aligned}$$

Then P_7 has 5 distinct stratifications, namely,

$$\begin{aligned} &\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\} \\ &\{r_2\}, \{r_1\}, \{r_3\}, \{r_4\} \\ &\{r_2\}, \{r_3\}, \{r_1\}, \{r_4\} \\ &\{r_1, r_2\}, \{r_3\}, \{r_4\} \\ &\{r_2\}, \{r_1, r_3\}, \{r_4\}. \end{aligned}$$

These lead to five different ways of reading the program P_7. As will be seen, each of these yields the same semantics.

There is a simple test for checking if a program is stratifiable. Not surprisingly, it involves testing for an acyclicity condition in definitions of relations using negation. Let P be a datalog$^\neg$ program. The *precedence graph* G_P of P is the labeled graph whose nodes are the *idb* relations of P. Its edges are the following:

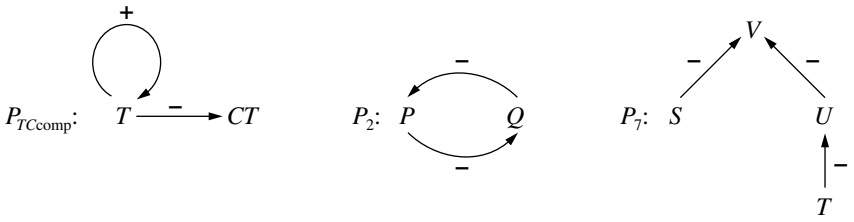


Figure 15.1: Precedence graphs for P_{CT}, P_2, and P_7

- If $R(u) \leftarrow \ldots R'(v) \ldots$ is a rule in P, then $\langle R', R\rangle$ is an edge in G_P with label $+$ (called a *positive edge*).
- If $R(u) \leftarrow \ldots \neg R'(v) \ldots$ is a rule in P, then $\langle R', R\rangle$ is an edge in G_P with label $-$ (called a *negative edge*).

For example, the precedence graphs for program $P_{TC\,comp}$, P_2, and P_7 are represented in Fig. 15.1. It is straightforward to show the following (proof omitted):

Lemma 15.2.6 Let P be a program with stratification σ. If there is a path from R' to R in G_P, then $\sigma(R') \leq \sigma(R)$; and if there is a path from R' to R in G_P containing some negative edge, then $\sigma(R') < \sigma(R)$.

We now show how the precedence graph of a program can be used to test the stratifiability of the program.

Proposition 15.2.7 A datalog$^\neg$ program P is stratifiable iff its precedence graph G_P has no cycle containing a negative edge.

Proof Consider the "only if" part. Suppose P is a datalog$^\neg$ program whose precedence graph has a cycle $R_1, \ldots R_m, R_1$ containing a negative edge, say from R_m to R_1. Suppose, toward a contradiction, that σ is a stratification mapping for P. By Lemma 15.2.6, $\sigma(R_1) < \sigma(R_1)$, because there is a path from R_1 to R_1 with a negative edge. This is a contradiction, so no stratification mapping σ exists for P.

Conversely, suppose P is a program whose precedence graph G_P has no cycle with negative edges. Let $\prec$ be the binary relation among the strongly connected components of G_P defined as follows: $C \prec C'$ if $C \neq C'$ and there is a (positive or negative) edge in G_P from some node of C to some node of C'.

We first show that

(*) $\prec$ is acyclic.

Suppose there is a cycle in $\prec$. Then by construction of $\prec$, this cycle must traverse two *distinct* strongly connected components, say C, C'. Let A be in C. It is easy to deduce that there is a path in G_P from some vertex in C' to A and from A to some vertex in C'.

Because C' is a strongly connected component of G_P, A is in C'. Thus $C \subseteq C'$, so $C = C'$, a contradiction. Hence (*) holds.

In view of (*), the binary relation $\prec$ induces a partial order among the strongly connected components of G_P, which we also denote by $\prec$, by abuse of notation. Let $C^1, \ldots, C^n$ be a topographic sort with respect to $\prec$ of the strongly connected components of G_P; that is, $C^1 \ldots C^n$ is the set of strongly connected components of G_P and if $C^i \prec C^j$, then $i \leq j$. Finally, for each i, $1 \leq i \leq n$, let Q^i consist of all rules defining some relation in C^i. Then $Q^1, \ldots, Q^n$ is a stratification of P. Indeed, (i) and (ii) in the definition of stratification are clearly satisfied. Conditions (iii) and (iv) follow immediately from the construction of G_P and $\prec$ and from the hypothesis that G_P has no cycle with negative edge. ■

Clearly, the stratifiability test provided by Proposition 15.2.7 takes time polynomial in the size of the program P.

Verification of the following observation is left to the reader (Exercise 15.4).

Lemma 15.2.8 Let $P^1, \ldots, P^n$ be a stratification of P, and let $Q^1, \ldots, Q^m$ be obtained as in Proposition 15.2.7. If $Q^j \cap P^i \neq \emptyset$, then $Q^j \subseteq P^i$. In particular, the partition $Q^1, \ldots, Q^m$ of P refines all other partitions given by stratifications of P.

Semantics of Stratified Programs

Consider a stratifiable program P with a stratification $\sigma = P^1, \ldots, P^n$. Using the stratification σ, we can now easily give a semantics to P using the well-understood semipositive programs. Notice that for each program P^i in the stratification, if P^i uses the negation of R', then $R' \in edb(P^i)$ [note that $edb(P^i)$ may contain some of the *idb* relations of P]. Furthermore, R' is either in $edb(P)$ or is defined by some P^j preceding P^i [i.e., $R' \in \cup_{j<i} idb(P^j)$]. Thus each program P^i is semipositive relative to previously defined relations. Then the semantics of P is obtained by applying, in order, the programs P^i. More precisely, let $\mathbf{I}$ be an instance over $edb(P)$. Define the sequence of instances

$$\mathbf{I}_0 = \mathbf{I}$$
$$\mathbf{I}_i = \mathbf{I}_{i-1} \cup P^i(\mathbf{I}_{i-1}|edb(P^i)), 0 < i \leq n.$$

Note that $\mathbf{I}_i$ extends $\mathbf{I}_{i-1}$ by providing values to the relations defined by P^i; and that $P^i(\mathbf{I}_{i-1}|edb(P^i))$, or equivalently, $P^i(\mathbf{I}_{i-1})$, is the semantics of the semipositive program P^i applied to the values of its *edb* relations provided by $\mathbf{I}_{i-1}$. Let us denote the final instance $\mathbf{I}_n$ thus obtained by $\sigma(\mathbf{I})$. This provides the *semantics of* a datalog$^\neg$ program *under a stratification* σ.

Independence of Stratification

As shown in Example 15.2.5, a datalog$^\neg$ program can have more than one stratification. Will the different stratifications yield the same semantics? Fortunately, the answer is yes.

To demonstrate this, we use the following simple lemma, whose proof is left to the reader (Exercise 15.10).

LEMMA 15.2.9 Let P be a semipositive datalog$^\neg$ program and σ a stratification for P. Then $P^{semi-pos}(\mathbf{I}) = \sigma(\mathbf{I})$ for each instance $\mathbf{I}$ over $edb(P)$.

Two stratifications of a datalog$^\neg$ program are *equivalent* if they yield the same semantics on all inputs.

THEOREM 15.2.10 Let P be a stratifiable datalog$^\neg$ program. All stratifications of P are equivalent.

Proof Let G_P be the precedence graph of P and $\sigma_{G_P} = Q^1, \ldots, Q^n$ be a stratification constructed from G_P as in the proof of Theorem 15.2.7. Let $\sigma = P^1, \ldots, P^k$ be a stratification of P. It clearly suffices to show that σ is equivalent to σ_{G_P}. The stratification σ_{G_P} is used as a reference because, as shown in Lemma 15.2.8, its strata are the finest possible among all stratifications for P.

As in the proof of Theorem 15.2.7, we use the partial order $\prec$ among the strongly connected components of G_P and the notation introduced there. Clearly, the relation $\prec$ on the C^i induces a partial order on the Q^i, which we also denote by $\prec$ ($Q^i \prec Q^j$ if $C^i \prec C^j$). We say that a sequence $Q^{i_1}, \ldots, Q^{i_r}$ of some of the Q^i is *compatible* with $\prec$ if for every $l < m$ it is *not* the case that $Q^{i_m} \prec Q^{i_l}$.

We shall prove that

1. If σ' and σ'' are permutations of σ_{G_P} that are compatible with $\prec$, then σ' and σ'' are equivalent stratifications of P.
2. For each P^i, $1 \leq i \leq k$, there exists $\sigma_i = Q^{i_1}, \ldots, Q^{i_r}$ such that σ_i is a stratification of P^i, and the sequence $Q^{i_1}, \ldots, Q^{i_r}$ is compatible with $\prec$.
3. $\sigma_1, \ldots, \sigma_k$ is a permutation of $Q^1, \ldots, Q^n$ compatible with $\prec$.

Before demonstrating these, we argue that the foregoing statements (1 through 3) are sufficient to show that σ and σ_{G_P} are equivalent. By statement 2, each σ_i is a stratification of P^i. Lemma 15.2.9 implies that P^i is equivalent to σ_i. It follows that $\sigma = P^1, \ldots, P^k$ is equivalent to $\sigma_1, \ldots, \sigma_k$ which, by statement 3, is a permutation of σ_{G_P} compatible with $\prec$. Then $\sigma_1, \ldots, \sigma_k$ and σ_{G_P} are equivalent by statement 1, so σ and σ_{G_P} are equivalent.

Consider statement 1. Note first that one can obtain σ'' from σ' by a sequence of exchanges of adjacent Q^i, Q^j such that $Q^i \not\prec Q^j$ and $Q^j \not\prec Q^i$ (Exercise 15.9). Thus it is sufficient to show that for every such pair, Q^i, Q^j is equivalent to Q^j, Q^i. Because $Q^i \not\prec Q^j$ and $Q^j \not\prec Q^i$, it follows that no *idb* relation of Q^i occurs in Q^j and conversely. Then $Q^i \cup Q^j$ is a semipositive program [with respect to $edb(Q^i \cup Q^j)$] and both Q^i, Q^j and Q^j, Q^i are stratifications of $Q^i \cup Q^j$. By Lemma 15.2.9, Q^i, Q^j and Q^j, Q^i are both equivalent to $Q^i \cup Q^j$ (as a semipositive program), so Q^i, Q^j and Q^j, Q^i are equivalent.

Statement 2 follows immediately from Lemma 15.2.8.

Finally, consider statement 3. By statement 2, each σ_i is compatible with $\prec$. Thus it remains to be shown that, if Q^m occurs in σ_i, Q^l occurs in σ_j, and $i < j$, then $Q^l \not\prec Q^m$.

Note that Q^l is included in P^j, and Q^m is included in P^i. It follows that for all relations R defined by Q^m and R' defined by Q^l, $\sigma(R) < \sigma(R')$, where σ is the stratification function of $P^1, \ldots, P^k$. Hence $R' \not\prec R$ so $Q^l \not\prec Q^m$. ■

Thus all stratifications of a given stratifiable program are equivalent. This means that we can speak about the semantics of such a program independently of a particular stratification. Given a stratifiable datalog$^\neg$ program P and an input $\mathbf{I}$ over $edb(P)$, we shall take as the semantics of P on $\mathbf{I}$ the semantics $\sigma(\mathbf{I})$ of any stratification σ of P. This semantics, well defined by Theorem 15.2.10, is denoted by $P^{strat}(\mathbf{I})$. Clearly, $P^{strat}(\mathbf{I})$ can be computed in time polynomial with respect to $\mathbf{I}$.

Now that we have a well-defined semantics for stratified programs, we can verify that for semipositive programs, the semantics coincides with the semantics already introduced. If P is a semipositive datalog$^\neg$ program, then P is also stratifiable. By Lemma 15.2.9, $P^{semi-pos}$ and P^{strat} are equivalent.

Properties of Stratified Semantics

Stratified semantics has a procedural flavor because it is the result of an ordering of the rules, albeit implicit. What can we say about $P^{strat}(\mathbf{I})$ from a model-theoretic point of view? Rather pleasantly, $P^{strat}(\mathbf{I})$ is a minimal model of Σ_P containing $\mathbf{I}$. However, no precise characterization of stratified semantics in model-theoretic terms has emerged. Some model-theoretic properties of stratified semantics are established next.

Proposition 15.2.11 For each stratifiable datalog$^\neg$ program P and instance $\mathbf{I}$ over $edb(\mathbf{I})$,

(a) $P^{strat}(\mathbf{I})$ is a minimal model of Σ_P whose restriction to $edb(P)$ equals $\mathbf{I}$.

(b) $P^{strat}(\mathbf{I})$ is a minimal fixpoint of T_P whose restriction to $edb(P)$ equals $\mathbf{I}$.

Proof For part (a), let $\sigma = P^1, \ldots, P^n$ be a stratification of P and $\mathbf{I}$ an instance over $edb(P)$. We have to show that $P^{strat}(\mathbf{I})$ is a minimal model of Σ_P whose restriction to $edb(P)$ equals $\mathbf{I}$. Clearly, $P^{strat}(\mathbf{I})$ is a model of Σ_P whose restriction to $edb(P)$ equals $\mathbf{I}$. To prove its minimality, it is sufficient to show that, for each model $\mathbf{J}$ of Σ_P,

$$(**) \qquad \text{if } \mathbf{I} \subseteq \mathbf{J} \subseteq P^{strat}(\mathbf{I}) \text{ then } \mathbf{J} = P^{strat}(\mathbf{I}).$$

Thus suppose $\mathbf{I} \subseteq \mathbf{J} \subseteq P^{strat}(\mathbf{I})$. We prove by induction on k that

$$(\dagger) \qquad P^{strat}(\mathbf{I})|sch(\cup_{i \le k} P^i) = \mathbf{J}|sch(\cup_{i \le k} P^i)$$

for each k, $1 \le k \le n$. The equality of $P^{strat}(\mathbf{I})$ and $\mathbf{J}$ then follows from ($\dagger$) with $k = n$.

For $k = 1$, $edb(P^1) \subseteq edb(P)$ so

$$P^{strat}(\mathbf{I})|edb(P^1) = \mathbf{I}|edb(P^1) = \mathbf{J}|edb(P^1).$$

By the definition of stratified semantics and Theorem 15.2.2, $P^{strat}(\mathbf{I})|sch(P^1)$ is the

minimum model of Σ_{P^1} whose restriction to $edb(P^1)$ equals $P^{strat}(\mathbf{I})|edb(P^1)$. On the other hand, $\mathbf{J}|sch(P^1)$ is also a model of Σ_{P^1} whose restriction to $edb(P^1)$ equals $P^{strat}(\mathbf{I})|edb(P^1)$. From the minimality of $P^{strat}(\mathbf{I})|sch(P^1)$, it follows that

$$P^{strat}(\mathbf{I})|sch(P^1) \subseteq \mathbf{J}|sch(P^1).$$

From (**) it then follows that $P^{strat}(\mathbf{I})|sch(P^1) = \mathbf{J}|sch(P^1)$, which establishes (†) for $k = 1$. For the induction step, suppose (†) is true for k, $1 \leq k < n$. Then (†) for $k + 1$ is shown in the same manner as for the case $k = 1$. This proves (†) for $1 \leq k \leq n$. It follows that $P^{strat}(\mathbf{I})$ is a minimal model of Σ_P whose restriction to $edb(P)$ equals $\mathbf{I}$.

The proof of part (b) is left for Exercise 15.12. ■

There is another appealing property of stratified semantics that takes into account the syntax of the program in addition to purely model-theoretic considerations. This property is illustrated next.

Consider the two programs

$$P_5 = \{p \leftarrow \neg q\}$$
$$P_6 = \{q \leftarrow \neg p\}$$

From the perspective of classical logic, Σ_{P_5} and Σ_{P_6} are equivalent to each other and to $\{p \vee q\}$. However, T_{P_5} and T_{P_6} have different behavior: The unique fixpoint of T_{P_5} is $\{p\}$, whereas that of T_{P_6} is $\{q\}$. This is partially captured by the notion of "supported" as follows.

Let datalog$^\neg$ program P and input $\mathbf{I}$ be given. As with pure datalog, $\mathbf{J}$ is a model of P iff $\mathbf{J} \supseteq T_P(\mathbf{J})$. We say that $\mathbf{J}$ is a *supported* model if $\mathbf{J} \subseteq T_P(\mathbf{J})$ (i.e., if each fact in $\mathbf{J}$ is "justified" or supported by being the head of a ground instantiation of a rule in P whose body is all true in $\mathbf{J}$). (In the context of some input $\mathbf{I}$, we say that $\mathbf{J}$ is supported *relative* to $\mathbf{I}$ and the definition is modified accordingly.) This condition, which has both syntactic and semantic aspects, captures at least some of the spirit of the immediate consequence operator T_P. As suggested in Remark 15.1.2, its impact can be annulled by adding rules of the form $p \leftarrow p$.

The proof of the following is left to the reader (Exercise 15.13).

Proposition 15.2.12 For each stratifiable program P and instance $\mathbf{I}$ over $edb(P)$, $P^{strat}(\mathbf{I})$ is a supported model of P relative to $\mathbf{I}$.

We have seen that stratification provides an elegant and simple approach to defining semantics of datalog$^\neg$ programs. Nonetheless, it has two major limitations. First, it does not provide semantics to *all* datalog$^\neg$ programs. Second, stratified datalog$^\neg$ programs are not entirely satisfactory with regard to expressive power. From a computational point of view, they provide recursion and negation and are inflationary. Therefore, as discussed in Chapter 14, one might expect that they express the *fixpoint* queries. Unfortunately, stratified datalog$^\neg$ programs fall short of expressing all such queries, as will be shown in Section 15.4. Intuitively, this is because the stratification condition prohibits recursive

application of negation, whereas in other languages expressing *fixpoint* this computational restriction does not exist.

For these reasons, we consider another semantics for datalog$^\neg$ programs called *well founded*. As we shall see, this provides semantics to all datalog$^\neg$ programs and expresses all *fixpoint* queries. Furthermore, well-founded and stratified semantics agree on stratified datalog$^\neg$ programs.

15.3 Well-Founded Semantics

Well-founded semantics relies on a fundamental revision of our expectations of the answer to a datalog$^\neg$ program. So far, we required that the answer must provide information on the truth or falsehood of every fact. Well-founded semantics is based on the idea that a given program may not necessarily provide such information on all facts. Instead some facts may simply be indifferent to it, and the answer should be allowed to say that the truth value of those facts is *unknown*. As it turns out, relaxing expectations about the answer in this fashion allows us to provide a natural semantics for *all* datalog$^\neg$ programs. The price is that the answer is no longer guaranteed to provide total information.

Another aspect of this approach is that it puts negative and positive facts on a more equal footing. One can no longer assume that $\neg R(u)$ is true simply because $R(u)$ is not in the answer. Instead, both negative and positive facts must be inferred. To formalize this, we shall introduce 3-valued instances, in which the truth value of facts can be **true**, **false**, or **unknown**.

This section begins by introducing a largely declarative semantics for datalog$^\neg$ programs. Next an equivalent fixpoint semantics is developed. Finally it is shown that stratified and well-founded semantics agree on the family of stratified datalog$^\neg$ programs.

A Declarative Semantics for Datalog$^\neg$

The aim of giving semantics to a datalog$^\neg$ program P will be to find an appropriate 3-valued model $\mathbf{I}$ of Σ_P. In considering what *appropriate* might mean, it is useful to recall the basic motivation underlying the logic-programming approach to negation as opposed to the purely computational approach. An important goal is to model some form of natural reasoning process. In particular, consistency in the reasoning process is required. Specifically, one cannot use a fact and later infer its negation. This should be captured in the notion of appropriateness of a 3-valued model $\mathbf{I}$, and it has two intuitive aspects:

- the positive facts of $\mathbf{I}$ must be inferred from P assuming the negative facts in $\mathbf{I}$; and
- all negative facts that can be inferred from $\mathbf{I}$ must already be in $\mathbf{I}$.

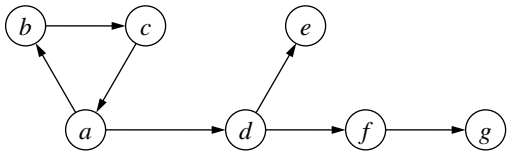
A 3-valued model satisfying the aforementioned notion of appropriateness will be called a 3-stable model of P. It turns out that, generally, programs have several 3-stable models. Then it is natural to take as an answer the certain (positive and negative) facts that belong to all such models, which turns out to yield, in some sense, the smallest 3-stable model. This is indeed how the well-founded semantics of P will be defined.

EXAMPLE 15.3.1 The example concerns a game with states, $a, b, \ldots$. The game is between two players. The possible moves of the games are held in a binary relation *moves*. A tuple $\langle a, b\rangle$ in *moves* indicates that when in state a, one can choose to move to state b. A player loses if he or she is in a state from which there are no moves. The goal is to compute the set of winning states (i.e., the set of states such that there exists a winning strategy for a player in this state). These are obtained in a unary predicate *win*.

Consider the input **K** with the following value for *moves*:

$$\mathbf{K}(moves) = \{\langle b, c\rangle, \langle c, a\rangle, \langle a, b\rangle, \langle a, d\rangle, \langle d, e\rangle, \langle d, f\rangle, \langle f, g\rangle\}$$

Graphically, the input is represented as



It is seen easily that there are indeed winning strategies from states d (move to e) and f (move to g). Slightly more subtle is the fact that there is no winning strategy from any of states a, b, or c. A given player can prevent the other from winning, essentially by forcing a nonterminating sequence of moves.

Now consider the following nonstratifiable program P_{win}:

$$win(x) \leftarrow moves(x, y), \neg win(y)$$

Intuitively, P_{win} states that a state x is in *win* if there is at least one state y that one can move to from x, for which the opposing player loses. We now exhibit a 3-valued model **J** of P_{win} that agrees with **K** on *moves*. As will be seen, this will in fact be the well-founded semantics of P_{win} on input **K**. Instance **J** is such that $\mathbf{J}(moves) = \mathbf{K}(moves)$ and the values of *win*-atoms are given as follows:

true	$win(d), win(f)$
false	$win(e), win(g)$
unknown	$win(a), win(b), win(c)$

We now embark on defining formally the well-founded semantics. We do this in three steps. First we define the notion of 3-valued instance and extend the notion of truth value and satisfaction. Then we consider datalog and show the existence of a minimum 3-valued model for each datalog program. Finally we consider datalog$^{\neg}$ and the notion of 3-stable model, which is the basis of well-founded semantics.

3-valued Instances Dealing with three truth values instead of the usual two requires extending some of the basic notions like instance and model. As we shall see, this is straightforward. We will denote **true** by 1, **false** by 0, and **unknown** by 1/2.

Consider a datalog$^\neg$ program P and a classical 2-valued instance $\mathbf{I}$. As was done in the discussion of SLD resolution in Chapter 12, we shall denote by $P_\mathbf{I}$ the program obtained from P by adding to P unit clauses stating that the facts in $\mathbf{I}$ are true. Then $P(\mathbf{I}) = P_\mathbf{I}(\emptyset)$. For the moment, we shall deal with datalog$^\neg$ programs such as these, whose input is included in the program. Recall that $\mathbf{B}(P)$ denotes all facts of the form $R(a_1, \ldots, a_k)$, where R is a relation and $a_1, \ldots, a_k$ constants occurring in P. In particular, $\mathbf{B}(P_\mathbf{I}) = \mathbf{B}(P, \mathbf{I})$.

Let P be a datalog$^\neg$ program. A *3-valued instance* $\mathbf{I}$ over $sch(P)$ is a total mapping from $\mathbf{B}(P)$ to $\{0, 1/2, 1\}$. We denote by $\mathbf{I}^1$, $\mathbf{I}^{1/2}$, and $\mathbf{I}^0$ the set of atoms in $\mathbf{B}(P)$ whose truth value is 1, $1/2$, and 0, respectively. A 3-valued instance $\mathbf{I}$ is *total*, or *2-valued*, if $\mathbf{I}^{1/2} = \emptyset$. There is a natural ordering $\prec$ among 3-valued instances over $sch(P)$, defined by

$$\mathbf{I} \prec \mathbf{J} \text{ iff for each } A \in \mathbf{B}(P),\ \ \mathbf{I}(A) \leq \mathbf{J}(A).$$

Note that this is equivalent to $\mathbf{I}^1 \subseteq \mathbf{J}^1$ and $\mathbf{I}^0 \supseteq \mathbf{J}^0$ and that it generalizes containment for 2-valued instances.

Occasionally, we will represent a 3-valued instance by listing the positive and negative facts and omitting the undefined ones. For example, the 3-valued instance $\mathbf{I}$, where $\mathbf{I}(p) = 1, \mathbf{I}(q) = 1, \mathbf{I}(r) = 1/2, \mathbf{I}(s) = 0$, will also be written as $\mathbf{I} = \{p, q, \neg s\}$.

Given a 3-valued instance $\mathbf{I}$, we next define the truth value of Boolean combinations of facts using the connectives $\vee, \wedge, \neg, \leftarrow$. The truth value of a Boolean combination α of facts is denoted by $\hat{\mathbf{I}}(\alpha)$, defined by

$$\begin{aligned}
\hat{\mathbf{I}}(\beta \wedge \gamma) &= \min\{\hat{\mathbf{I}}(\beta), \hat{\mathbf{I}}(\gamma)\} \\
\hat{\mathbf{I}}(\beta \vee \gamma) &= \max\{\hat{\mathbf{I}}(\beta), \hat{\mathbf{I}}(\gamma)\} \\
\hat{\mathbf{I}}(\neg\beta) &= 1 - \hat{\mathbf{I}}(\beta) \\
\hat{\mathbf{I}}(\beta \leftarrow \gamma) &= 1 \text{ if } \hat{\mathbf{I}}(\gamma) \leq \hat{\mathbf{I}}(\beta), \text{ and 0 otherwise.}
\end{aligned}$$

The reader should be careful: Known facts about Boolean operators in the 2-valued context may not hold in this more complex one. For instance, note that the truth value of $p \leftarrow q$ may be different from that of $p \vee \neg q$ (see Exercise 15.15). To see that the preceding definition matches the intuition, one might want to verify that with the specific semantics of $\leftarrow$ used here, the instance $\mathbf{J}$ of Example 15.3.1 does satisfy (the ground instantiation of) $P_{win,\mathbf{K}}$. That would not be the case if we define the semantics of $\leftarrow$ in a more standard way; by using $p \leftarrow q \equiv p \vee \neg q$.

A 3-valued instance $\mathbf{I}$ over $sch(P)$ *satisfies* a Boolean combination α of atoms in $\mathbf{B}(P)$ iff $\hat{\mathbf{I}}(\alpha) = 1$. Given a datalog$^{(\neg)}$ program P, a *3-valued model* of Σ_P is a 3-valued instance over $sch(P)$ satisfying the set of implications corresponding to the rules in $ground(P)$.

Example 15.3.2 Recall the program P_{win} of Example 15.3.1 and the input instance $\mathbf{K}$ and output instance $\mathbf{J}$ presented there. Consider these ground sentences:

$$win(a) \leftarrow moves(a, d), \neg win(d)$$
$$win(a) \leftarrow moves(a, b), \neg win(b).$$

The first is true for $\mathbf{J}$, because $\hat{\mathbf{J}}(\neg win(d)) = 0$, $\hat{\mathbf{J}}(moves(a, d)) = 1$, $\hat{\mathbf{J}}(win(a)) = 1/2$, and $1/2 \geq 0$. The second is true because $\hat{\mathbf{J}}(\neg win(b)) = 1/2$, $\hat{\mathbf{J}}(moves(a, b)) = 1$, $\hat{\mathbf{J}}(win(a)) = 1/2$, and $1/2 \geq 1/2$.

Observe that, on the other hand,

$$\hat{\mathbf{J}}(win(a) \vee \neg(moves(a, b) \wedge \neg win(b))) = 1/2.$$

3-valued Minimal Model for Datalog We next extend the definition and semantics of datalog programs to the context of 3-valued instances. Although datalog programs do not contain negation, they will now be allowed to infer positive, unknown, and false facts. The syntax of a *3-extended datalog program* is the same as for datalog, except that the truth values $0, 1/2$, and 1 can occur as literals in bodies of rules. Given a 3-extended datalog program P, the *3-valued immediate consequence operator* $3\text{-}T_P$ of P is a mapping on 3-valued instances over $sch(P)$ defined as follows. Given a 3-valued instance $\mathbf{I}$ and $A \in \mathbf{B}(P)$, $3\text{-}T_P(\mathbf{I})(A)$ is

1 if there is a rule $A \leftarrow body$ in $ground(P)$ such that $\hat{\mathbf{I}}(body) = 1$,

0 if for each rule $A \leftarrow body$ in $ground(P)$, $\hat{\mathbf{I}}(body) = 0$ (and, in particular, if there is no rule with A in head),

$1/2$ otherwise.

Example 15.3.3 Consider the 3-extended datalog program $P = \{p \leftarrow 1/2;\ p \leftarrow q, 1/2;\ q \leftarrow p, r;\ q \leftarrow p, s;\ s \leftarrow q;\ r \leftarrow 1\}$. Then

$$\begin{aligned} 3\text{-}T_P(\{\neg p, \neg q, \neg r, \neg s\}) &= \{\neg q, r, \neg s\} \\ 3\text{-}T_P(\{\neg q, r, \neg s\}) &= \{r, \neg s\} \\ 3\text{-}T_P(\{r, \neg s\}) &= \{r\} \\ 3\text{-}T_P(\{r\}) &= \{r\}. \end{aligned}$$

In the following, 3-valued instances are compared with respect to $\prec$. Thus "least," "minimal," and "monotonic" are with respect to $\prec$ rather than the set inclusion used for classical 2-valued instances. In particular, note that the minimum 3-valued instance with respect to $\prec$ is that where all atoms are false. Let $\perp$ denote this particular instance.

With the preceding definitions, extended datalog programs on 3-valued instances behave similarly to classical programs. The next lemma can be verified easily (Exercise 15.16):

Lemma 15.3.4 Let P be a 3-extended datalog program. Then

1. $3{-}T_P$ is monotonic and the sequence $\{3\text{-}T_P^i(\perp)\}_{i>0}$ is increasing and converges to the least fixpoint of $3\text{-}T_P$;

2. P has a unique minimal 3-valued model that equals the least fixpoint of $3\text{-}T_P$.

The semantics of an extended datalog program is the minimum 3-valued model of P. Analogous to conventional datalog, we denote this by $P(\perp)$.

3-stable Models of Datalog$^\neg$

We are now ready to look at datalog$^\neg$ programs and formally define 3-stable models of a datalog$^\neg$ program P. We "bootstrap" to the semantics of programs with negation, using the semantics for 3-extended datalog programs described earlier. Let $\mathbf{I}$ be a 3-valued instance over $sch(P)$. We reduce the problem to that of applying a positive datalog program, as follows. The *positivized ground version* of P given $\mathbf{I}$, denoted $pg(P, \mathbf{I})$, is the 3-extended datalog program obtained from $ground(P)$ by replacing each negative premise $\neg A$ by $\hat{\mathbf{I}}(\neg A)$ (i.e., 0, 1, or 1/2). Because all negative literals in $ground(P)$ have been replaced by their truth value in $\mathbf{I}$, $pg(P, \mathbf{I})$ is now a 3-extended datalog program (i.e, a program without negation). Its least fixpoint $pg(P, \mathbf{I})(\perp)$ contains all the facts that are consequences of P by assuming the values for the negative premises as given by $\mathbf{I}$. We denote $pg(P, \mathbf{I})(\perp)$ by $conseq_P(\mathbf{I})$. Thus the intuitive conditions required of 3-stable models now amount to $conseq_P(\mathbf{I}) = \mathbf{I}$.

DEFINITION 15.3.5 Let P be a datalog$^\neg$ program. A 3-valued instance $\mathbf{I}$ over $sch(P)$ is a *3-stable model* of P iff $conseq_P(\mathbf{I}) = \mathbf{I}$.

Observe an important distinction between $conseq_P$ and the immediate consequence operator used for inflationary datalog$^\neg$. For inflationary datalog$^\neg$, we assumed that $\neg A$ was true as long as A was not inferred. Here we just assume in such a case that A is unknown and try to prove new facts. Of course, doing so requires the 3-valued approach.

EXAMPLE 15.3.6 Consider the following datalog$^\neg$ program P:

$$
\begin{aligned}
p &\leftarrow \neg r \\
q &\leftarrow \neg r,\ p \\
s &\leftarrow \neg t \\
t &\leftarrow q,\ \neg s \\
u &\leftarrow \neg t,\ p,\ s
\end{aligned}
$$

The program has three 3-stable models (represented by listing the positive and negative facts and leaving out the unknown facts):

$$
\begin{aligned}
\mathbf{I}_1 &= \{p, q, t, \neg r, \neg s, \neg u\} \\
\mathbf{I}_2 &= \{p, q, s, \neg r, \neg t, \neg u\} \\
\mathbf{I}_3 &= \{p, q, \neg r\}
\end{aligned}
$$

Let us check that $\mathbf{I}_3$ is a 3-stable model of P. The program $P' = pg(P, \mathbf{I}_3)$ is

$$p \leftarrow 1$$
$$q \leftarrow 1, p$$
$$s \leftarrow 1/2$$
$$t \leftarrow q, 1/2$$
$$u \leftarrow 1/2, p, s$$

The minimum 3-valued model of $pg(P, \mathbf{I}_3)$ is obtained by iterating $3\text{-}T_{P'}(\bot)$ up to a fixpoint. Thus we start with $\bot = \{\neg p, \neg q, \neg r, \neg s, \neg t, \neg u\}$. The first application of $3\text{-}T_{P'}$ yields $3\text{-}T_{P'}(\bot) = \{p, \neg q, \neg r, \neg t, \neg u\}$. Next $(3\text{-}T_{P'})^2(\bot) = \{p, q, \neg r, \neg t\}$. Finally $(3\text{-}T_{P'})^3(\bot) = (3\text{-}T_{P'})^4(\bot) = \{p, q, \neg r\}$. Thus

$$conseq_P(\mathbf{I}_3) = pg(P, \mathbf{I}_3)(\bot) = (3\text{-}T_{P'})^3(\bot) = \mathbf{I}_3,$$

and $\mathbf{I}_3$ is a 3-stable model of P.

The reader is invited to verify that in Example 15.3.1, the instance $\mathbf{J}$ is a 3-stable model of the program $P_{win,\mathbf{K}}$ for the input instance $\mathbf{K}$ presented there.

As seen from the example, datalog$^\neg$ programs generally have several 3-stable models. We will show later that each datalog$^\neg$ program has at least one 3-stable model. Therefore it makes sense to let the final answer consist of the positive and negative facts belonging to all 3-stable models of the program. As we shall see, the 3-valued instance so obtained is itself a 3-stable model of the program.

DEFINITION 15.3.7 Let P be a datalog$^\neg$ program. The *well-founded semantics* of P is the 3-valued instance consisting of all positive and negative facts belonging to all 3-stable models of P. This is denoted by $P^{wf}(\emptyset)$,or simply, P^{wf}. Given datalog$^\neg$ program P and input instance $\mathbf{I}$, $P^{wf}_{\mathbf{I}}(\emptyset)$ is denoted $P^{wf}(\mathbf{I})$.

Thus the well-founded semantics of the program P in Example 15.3.6 is $P^{wf}(\emptyset) = \{p, q, \neg r\}$. We shall see later that in Example 15.3.1, $P^{wf}_{win}(\mathbf{K}) = \mathbf{J}$.

A Fixpoint Definition

Note that the preceding description of the well-founded semantics, although effective, is inefficient. The straightforward algorithm yielded by this description involves checking all possible 3-valued instances of a program, determining which are 3-stable models, and then taking their intersection. We next provide a simpler, efficient way of computing the well-founded semantics. It is based on an "alternating fixpoint" computation that converges to the well-founded semantics. As a side-effect, the proof will show that each datalog$^\neg$ program has at least one 3-stable model (and therefore the well-founded semantics is always defined), something we have not proven. It will also show that the well-founded model is itself a 3-stable model, in some sense the smallest.

The idea of the computation is as follows. We define an alternating sequence $\{\mathbf{I}_i\}_{i \geq 0}$ of 3-valued instances that are underestimates and overestimates of the facts known in every

3-stable model of P. The sequence is as follows:

$$\begin{aligned}\mathbf{I}_0 &= \bot \\ \mathbf{I}_{i+1} &= conseq_P(\mathbf{I}_i).\end{aligned}$$

Recall that $\bot$ is the least 3-valued instance and that all facts have value 0 in $\bot$. Also note that each of the $\mathbf{I}_i$ just defined is a total instance. This follows easily from the following facts (Exercise 15.17):

- if $\mathbf{I}$ is total, then $conseq_P(\mathbf{I})$ is total; and
- the $\mathbf{I}_i$ are constructed starting from the total instance $\bot$ by repeated applications of $conseq_P$.

The intuition behind the construction of the sequence $\{\mathbf{I}_i\}_{i\geq 0}$ is the following. The sequence starts with $\bot$, which is an overestimate of the negative facts in the answer (it contains all negative facts). From this overestimate we compute $\mathbf{I}_1 = conseq_P(\bot)$, which includes all positive facts that can be inferred from $\bot$. This is clearly an overestimate of the positive facts in the answer, so the set of negative facts in $\mathbf{I}_1$ is an underestimate of the negative facts in the answer. Using this underestimate of the negative facts, we compute $\mathbf{I}_2 = conseq_P(\mathbf{I}_1)$, whose positive facts will now be an underestimate of the positive facts in the answer. By continuing the process, we see that the even-indexed instances provide underestimates of the positive facts in the answer and the odd-indexed ones provide underestimates of the negative facts in the answer. Then the limit of the even-indexed instances provides the positive facts in the answer and the limit of the odd-indexed instances provides the negative facts in the answer. This intuition will be made formal later in this section.

It is easy to see that $conseq_P(\mathbf{I})$ is antimonotonic. That is, if $\mathbf{I} \prec \mathbf{J}$, then $conseq_P(\mathbf{J}) \prec conseq_P(\mathbf{I})$ (Exercise 15.17). From this and the facts that $\bot \prec \mathbf{I}_1$ and $\bot \prec \mathbf{I}_2$, it immediately follows that, for all $i > 0$,

$$\mathbf{I}_0 \prec \mathbf{I}_2 \ldots \prec \mathbf{I}_{2i} \prec \mathbf{I}_{2i+2} \prec \ldots \prec \mathbf{I}_{2i+1} \prec \mathbf{I}_{2i-1} \prec \ldots \prec \mathbf{I}_1.$$

Thus the even subsequence is increasing and the odd one is decreasing. Because there are finitely many 3-valued instances relative to a given program P, each of these sequences becomes constant at some point. Let $\mathbf{I}_*$ denote the limit of the increasing sequence $\{\mathbf{I}_{2i}\}_{i\geq 0}$, and let $\mathbf{I}^*$ denote the limit of the decreasing sequence $\{\mathbf{I}_{2i+1}\}_{i\geq 0}$. From the aforementioned inequalities, it follows that $\mathbf{I}_* \prec \mathbf{I}^*$. Moreover, note that $conseq_P(\mathbf{I}_*) = \mathbf{I}^*$ and $conseq_P(\mathbf{I}^*) = \mathbf{I}_*$. Finally let $\mathbf{I}^*_*$ denote the 3-valued instance consisting of the facts known in both $\mathbf{I}_*$ and $\mathbf{I}^*$; that is,

$$\mathbf{I}^*_*(A) = \begin{cases} 1 & \text{if } \mathbf{I}_*(A) = \mathbf{I}^*(A) = 1 \\ 0 & \text{if } \mathbf{I}_*(A) = \mathbf{I}^*(A) = 0 \text{ and} \\ 1/2 & \text{otherwise.} \end{cases}$$

Equivalently, $\mathbf{I}^*_* = (\mathbf{I}_*)^1 \cup (\mathbf{I}^*)^0$. As will be seen shortly, $\mathbf{I}^*_* = P^{wf}(\emptyset)$. Before proving this, we illustrate the alternating fixpoint computation with several examples.

EXAMPLE 15.3.8

(a) Consider again the program in Example 15.3.6. Let us perform the alternating fixpoint computation described earlier. We start with $\mathbf{I}_0 = \bot = \{\neg p, \neg q, \neg r, \neg s, \neg t, \neg u\}$. By applying $conseq_P$, we obtain the following sequence of instances:

$$\mathbf{I}_1 = \{p, q, \neg r, s, t, u\},$$
$$\mathbf{I}_2 = \{p, q, \neg r, \neg s, \neg t, \neg u\},$$
$$\mathbf{I}_3 = \{p, q, \neg r, s, t, u\},$$
$$\mathbf{I}_4 = \{p, q, \neg r, \neg s, \neg t, \neg u\}.$$

Thus $\mathbf{I}_* = \mathbf{I}_4 = \{p, q, \neg r, \neg s, \neg t, \neg u\}$ and $\mathbf{I}^* = \mathbf{I}_3 = \{p, q, \neg r, s, t, u\}$. Finally $\mathbf{I}^*_* = \{p, q, \neg r\}$, which coincides with the well-founded semantics of P computed in Example 15.3.6.

(b) Recall now P_{win} and input $\mathbf{K}$ of Example 15.3.1. We compute $\mathbf{I}^*_*$ for the program $P_{win,\mathbf{I}}$. Note that for $\mathbf{I}_0$ the value of all *move* atoms is **false**, and for each $j \geq 1$, $\mathbf{I}_j$ agrees with the input $\mathbf{K}$ on the predicate *moves*; thus we do not show the *move* atoms here. For the *win* predicate, then, we have

$$\mathbf{I}_1 = \{win(a), win(b), win(c), win(d), \neg win(e), win(f), \neg win(g)\}$$
$$\mathbf{I}_2 = \{\neg win(a), \neg win(b), \neg win(c), win(d), \neg win(e), win(f), \neg win(g)\}$$
$$\mathbf{I}_3 = \mathbf{I}_1$$
$$\mathbf{I}_4 = \mathbf{I}_2.$$

Thus

$$\mathbf{I}_* = \mathbf{I}_2 = \{\neg win(a), \neg win(b), \neg win(c), win(d), \neg win(e), win(f), \neg win(g)\}$$
$$\mathbf{I}^* = \mathbf{I}_1 = \{win(a), win(b), win(c), win(d), \neg win(e), win(f), \neg win(g)\}$$
$$\mathbf{I}^*_* = \{win(d), \neg win(e), win(f), \neg win(g)\},$$

which is the instance $\mathbf{J}$ of Example 15.3.1.

(c) Consider the database schema consisting of a binary relation G and a unary relation *good* and the following program defining *bad* and *answer*:

$$bad(x) \leftarrow G(y, x), \neg good(y)$$
$$answer(x) \leftarrow \neg bad(x)$$

Consider the instance $\mathbf{K}$ over G and *good*, where

$$\mathbf{K}(G) = \{\langle b, c\rangle, \langle c, b\rangle, \langle c, d\rangle, \langle a, d\rangle, \langle a, e\rangle\}, \text{ and}$$
$$\mathbf{K}(good) = \{\langle a\rangle\}.$$

We assume that the facts of the database are added as unit clauses to P, yielding $P_{\mathbf{K}}$. Again we perform the alternating fixpoint computation for $P_{\mathbf{K}}$. We start with

$\mathbf{I}_0 = \perp$ (containing all negated atoms). Applying $conseq_{P_\mathbf{K}}$ yields the following sequence $\{\mathbf{I}_i\}_{i>0}$:

	bad	*answer*
$\mathbf{I}_0$	$\emptyset$	$\emptyset$
$\mathbf{I}_1$	$\{\neg a, b, c, d, e\}$	$\{a, b, c, d, e\}$
$\mathbf{I}_2$	$\{\neg a, b, c, d, \neg e\}$	$\{a, \neg b, \neg c, \neg d, \neg e\}$
$\mathbf{I}_3$	$\{\neg a, b, c, d, \neg e\}$	$\{a, \neg b, \neg c, \neg d, e\}$
$\mathbf{I}_4$	$\{\neg a, b, c, d, \neg e\}$	$\{a, \neg b, \neg c, \neg d, e\}$

We have omitted [as in (b)] the facts relating to the *edb* predicates G and *good*, which do not change after step 1.

Thus $\mathbf{I}^*_* = \mathbf{I}_* = \mathbf{I}^* = \mathbf{I}_3 = \mathbf{I}_4$. Note that P is stratified and its well-founded semantics coincides with its stratified semantics. As we shall see, this is not accidental.

We now show that the fixpoint construction yields the well-founded semantics for datalog$^\neg$ programs.

THEOREM 15.3.9 For each datalog$^\neg$ program P,

1. $\mathbf{I}^*_*$ is a 3-stable model of P.
2. $P^{wf}(\emptyset) = \mathbf{I}^*_*$.

Proof For statement 1, we need to show that $conseq_P(\mathbf{I}^*_*) = \mathbf{I}^*_*$. We show that for every fact A, if $\mathbf{I}^*_*(A) = \epsilon \in \{0, 1/2, 1\}$, then $conseq_P(\mathbf{I}^*_*)(A) = \epsilon$. From the antimonotonicity of $conseq_P$, the fact that $\mathbf{I}_* \prec \mathbf{I}^*_* \prec \mathbf{I}^*$ and $conseq_P(\mathbf{I}_*) = \mathbf{I}^*$, $conseq_P(\mathbf{I}^*) = \mathbf{I}_*$, it follows that $\mathbf{I}_* \prec conseq_P(\mathbf{I}^*_*) \prec \mathbf{I}^*$. If $\mathbf{I}^*_*(A) = 0$, then $\mathbf{I}^*(A) = 0$ so $conseq_P(\mathbf{I}^*_*)(A) = 0$; similarly for $\mathbf{I}^*_*(A) = 1$. Now suppose that $\mathbf{I}^*_*(A) = 1/2$. It is sufficient to prove that $conseq_P(\mathbf{I}^*_*)(A) \geq 1/2$. [It is not possible that $conseq_P(\mathbf{I}^*_*)(A) = 1$. If this were the case, the rules used to infer A involve only facts whose value is 0 or 1. Because those facts have the same value in $\mathbf{I}_*$ and $\mathbf{I}^*$, the same rules can be used in both $pg(P, \mathbf{I}_*)$ and $pg(P, \mathbf{I}^*)$ to infer A, so $\mathbf{I}_*(A) = \mathbf{I}^*(A) = \mathbf{I}^*_*(A) = 1$, which contradicts the hypothesis that $\mathbf{I}^*_*(A) = 1/2$.]

We now prove that $conseq_P(\mathbf{I}^*_*)(A) \geq 1/2$. By the definition of $\mathbf{I}^*_*$, $\mathbf{I}_*(A) = 0$ and $\mathbf{I}^*(A) = 1$. Recall that $conseq_P(\mathbf{I}_*) = \mathbf{I}^*$, so $conseq_P(\mathbf{I}_*)(A) = 1$. In addition, $conseq_P(\mathbf{I}_*)$ is the limit of the sequence $\{3\text{-}T^i_{pg(P,\mathbf{I}_*)}\}_{i>0}$. Let $stage(A)$ be the minimum i such that $3\text{-}T^i_{pg(P,\mathbf{I}_*)}(A) = 1$. We prove by induction on $stage(A)$ that $conseq_P(\mathbf{I}^*_*)(A) \geq 1/2$. Suppose that $stage(A) = 1$. Then there exists in $ground(P)$ a rule of the form $A \leftarrow$, or one of the form $A \leftarrow \neg B_1, \ldots, \neg B_n$, where $\mathbf{I}_*(B_j) = 0, 1 \leq j \leq n$. However, the first case cannot occur, for otherwise $conseq_P(\mathbf{I}^*)(A)$ must also equal 1 so $\mathbf{I}_*(A) = 1$ and therefore $\mathbf{I}^*_*(A) = 1$, contradicting the fact that $\mathbf{I}^*_*(A) = 1/2$. By the same argument, $\mathbf{I}^*(B_j) = 1$, so $\mathbf{I}^*_*(B_j) = 1/2, 1 \leq j \leq n$. Consider now $pg(P, \mathbf{I}^*_*)$. Because $\mathbf{I}^*_*(B_j) =$

$1/2$, $1 \leq j \leq n$, the second rule yields $conseq_P(\mathbf{I}^*_*)(A) \geq 1/2$. Now suppose that the statement is true for $stage(A) = i$ and suppose that $stage(A) = i + 1$. Then there exists a rule $A \leftarrow A_1 \ldots A_m \neg B_1 \ldots \neg B_n$ such that $\mathbf{I}_*(B_j) = 0$ and $3\text{-}T^i_{pg(P,\mathbf{I}_*)}(A_k) = 1$ for each j and k. Because $\mathbf{I}_*(B_j) = 0$, $\mathbf{I}^*_*(B_j) \leq 1/2$ so $\mathbf{I}^*_*(\neg B_j) \geq 1/2$. In addition, by the induction hypothesis, $conseq_P(\mathbf{I}^*_*)(A_k) \geq 1/2$. It follows that $conseq_P(\mathbf{I}^*_*)(A) \geq 1/2$, and the induction is complete. Thus $conseq_P(\mathbf{I}^*_*) = \mathbf{I}^*_*$ and $\mathbf{I}^*_*$ is a 3-stable model of P.

Consider statement 2. We have to show that the positive and negative facts in $\mathbf{I}^*_*$ are those belonging to every 3-stable model $\mathbf{M}$ of P. Because $\mathbf{I}^*_*$ is itself a 3-stable model of P, it contains the positive and negative facts belonging to every 3-stable model of P. It remains to show the converse (i.e., that the positive and negative facts in $\mathbf{I}^*_*$ belong to every 3-stable model of P). To this end, we first show that for each 3-stable model $\mathbf{M}$ of P and $i \geq 0$,

(‡) $$\mathbf{I}_{2i} \prec \mathbf{M} \prec \mathbf{I}_{2i+1}.$$

The proof is by induction on i. For $i = 0$, we have

$$\mathbf{I}_0 = \bot \prec \mathbf{M}.$$

Because $conseq_P$ is antimonotonic, $conseq_P(\mathbf{M}) \prec conseq_P(\mathbf{I}_0)$. Now $conseq_P(\mathbf{I}_0) = \mathbf{I}_1$ and because $\mathbf{M}$ is 3-stable, $conseq_P(\mathbf{M}) = \mathbf{M}$. Thus we have

$$\mathbf{I}_0 \prec \mathbf{M} \prec \mathbf{I}_1.$$

The induction step is similar and is omitted.

By (‡), $\mathbf{I}_* \prec \mathbf{M} \prec \mathbf{I}^*$. Now a positive fact in $\mathbf{I}^*_*$ is in $\mathbf{I}_*$ and so is in $\mathbf{M}$ because $\mathbf{I}_* \prec \mathbf{M}$. Similarly, a negative fact in $\mathbf{I}^*_*$ is in $\mathbf{I}^*$ and so is in $\mathbf{M}$ because $\mathbf{M} \prec \mathbf{I}^*$. ■

Note that the proof of statement 2 above formalizes the intuition that the $\mathbf{I}_{2i}$ provide underestimates of the positive facts in all acceptable answers (3-stable models) and the $\mathbf{I}_{2i+1}$ provide underestimates of the negative facts in those answers. The fact that $P^{wf}(\emptyset)$ is a minimal model of P is left for Exercise 15.19.

Variations of the alternating fixpoint computation can be obtained by starting with initial instances different from $\bot$. For example, it may make sense to start with the content of the *edb* relations as an initial instance. Such variations are sometimes useful for technical reasons. It turns out that the resulting sequences still compute the well-founded semantics. We show the following:

PROPOSITION 15.3.10 Let P be a datalog$^\neg$ program. Let $\{\bar{\mathbf{I}}_i\}_{i \geq 0}$ be defined in the same way as the sequence $\{\mathbf{I}_i\}_{i \geq 0}$, except that $\bar{\mathbf{I}}_0$ is some total instance such that

$$\bot \prec \bar{\mathbf{I}}_0 \prec P^{wf}(\emptyset).$$

Then

$$\bar{\mathbf{I}}_0 \prec \bar{\mathbf{I}}_2 \ldots \prec \bar{\mathbf{I}}_{2i} \prec \bar{\mathbf{I}}_{2i+2} \prec \ldots \prec \bar{\mathbf{I}}_{2i+1} \prec \bar{\mathbf{I}}_{2i-1} \prec \ldots \prec \bar{\mathbf{I}}_1$$

and (using the same notation as before),

$$\bar{\mathbf{I}}^*_* = P^{wf}(\emptyset).$$

Proof Let us compare the sequences $\{\mathbf{I}_i\}_{i\geq 0}$ and $\{\bar{\mathbf{I}}_i\}_{i\geq 0}$. Because $\bar{\mathbf{I}}_0 \prec P^{wf}(\emptyset)$ and $\bar{\mathbf{I}}_0$ is total, it easily follows that $\bar{\mathbf{I}}_0 \prec \mathbf{I}_*$. Thus $\perp = \mathbf{I}_0 \prec \bar{\mathbf{I}}_0 \prec \mathbf{I}_*$. From the antimonotonicity of the $conseq_P$ operator and the fact that $conseq^2_P(\mathbf{I}_*) = \mathbf{I}_*$, it follows that $\mathbf{I}_{2i} \prec \bar{\mathbf{I}}_{2i} \prec \mathbf{I}_*$ for all i, $i \geq 0$. Thus $\bar{\mathbf{I}}_* = \mathbf{I}_*$. Then

$$\bar{\mathbf{I}}^* = conseq_P(\bar{\mathbf{I}}_*) = conseq_P(\mathbf{I}_*) = \mathbf{I}^*$$

so $\bar{\mathbf{I}}^*_* = \mathbf{I}^*_* = P^{wf}(\emptyset)$. ■

As noted earlier, the instances in the sequence $\{\mathbf{I}_i\}_{i\geq 0}$ are total. A slightly different alternating fixpoint computation formulated only in terms of positive and negative facts can be defined. This is explored in Exercise 15.25.

Finally, the alternating fixpoint computation of the well-founded semantics involves looking at the ground rules of the given program. However, one can clearly compute the semantics without having to explicitly look at the ground rules. We show in Section 15.4 how the well-founded semantics can be computed by a *fixpoint* query.

Well-Founded and Stratified Semantics Agree

Because the well-founded semantics provides semantics to all datalog$^\neg$ programs, it does so in particular for stratified programs. Example 15.3.8(c) showed one stratified program for which stratified and well-founded semantics coincide. Fortunately, as shown next, stratified and well-founded semantics are always compatible. Thus if a program is stratified, then the stratified and well-founded semantics agree.

A datalog$^\neg$ program P is said to be *total* if $P^{wf}(\mathbf{I})$ is total for each input $\mathbf{I}$ over $edb(P)$.

Theorem 15.3.11 If P is a stratified datalog$^\neg$ program, then P is total under the well-founded semantics, and for each 2-valued instance $\mathbf{I}$ over $edb(P)$, $P^{wf}(\mathbf{I}) = P^{strat}(\mathbf{I})$.

Proof Let P be stratified, and let input $\mathbf{I}_0$ over $edb(P)$ be fixed. The idea of the proof is the following. Let $\mathbf{J}$ be a 3-stable model of $P_{\mathbf{I}_0}$. We shall show that $\mathbf{J} = P^{strat}(\mathbf{I}_0)$. This will imply that $P^{strat}(\mathbf{I}_0)$ is the unique 3-stable model for $P_{\mathbf{I}_0}$. In particular, it contains only the positive and negative facts in all 3-stable models of $P_{\mathbf{I}_0}$ and is thus $P^{wf}(\mathbf{I}_0)$.

For the proof, we will need to develop some notation.

Notation for the stratification: Let $P^1, \ldots, P^n$ be a stratification of P. Let $P^0 = \emptyset_{\mathbf{I}_0}$ (i.e., the program corresponding to all of the facts in $\mathbf{I}_0$). For each k in $[0, n]$,

- let $\mathbf{S}_k = idb(P^k)$ ($\mathbf{S}_0$ is $edb(P)$);
- $\mathbf{S}_{[0,k]} = \cup_{i\in[0,k]}\mathbf{S}_i$; and

▪ $\mathbf{I}_k = (P^1 \cup \cdots \cup P^k)^{strat}(\mathbf{I}_0) = \mathbf{I}_n|\mathbf{S}_{[0,k]}$ (and, in particular, $P^{strat}(\mathbf{I}_0) = \mathbf{I}_n$).

Notation for the 3-stable model: Let $\hat{P} = pg(P_{\mathbf{I}_0}, \mathbf{J})$. Recall that because $\mathbf{J}$ is 3-stable for $P_{\mathbf{I}_0}$,

$$\mathbf{J} = conseq_{\hat{P}}(\mathbf{J}) = \lim_{i \geq 0} 3\text{-}T^i_{\hat{P}}(\emptyset).$$

For each k in $[0, n]$,

- let $\mathbf{J}_k = \mathbf{J}|\mathbf{S}_{[0,k]}$; and
- $\hat{P}^{k+1} = pg(P^{k+1}_{\mathbf{J}_k}, \mathbf{J}_k) = pg(P^{k+1}_{\mathbf{J}_k}, \mathbf{J})$.

[Note that $pg(P^{k+1}_{\mathbf{J}_k}, \mathbf{J}_k) = pg(P^{k+1}_{\mathbf{J}_k}, \mathbf{J})$ because all the negations in P^{k+1} are over predicates in $\mathbf{S}_{[0,k]}$.]

To demonstrate the result, we will show by induction on $k \in [0, n]$ that

(*) $$\exists l_k \geq 0 \text{ such that } \forall i \geq 0, \mathbf{J}_k = 3\text{-}T^{l_k+i}_{\hat{P}}(\emptyset) \mid \mathbf{S}_{[0,k]} = \mathbf{I}_k.$$

Clearly, for $k = n$, (*) demonstrates the result.

The case where $k = 0$ is satisfied by setting $l_0 = 1$, because $\mathbf{J}_0 = 3\text{-}T^{1+i}_{\hat{P}}(\emptyset)|\mathbf{S}_0 = \mathbf{I}_0$ for each $i \geq 0$.

Suppose now that (*) is true for some $k \in [0, n-1]$. Then for each $i \geq 0$, by the choice of $\hat{P}^{k+1}$, the form of P^{k+1}, and (*),

(1) $$T^i_{P^{k+1}}(\mathbf{I}_k)|\mathbf{S}_{k+1} \subseteq 3\text{-}T^{i+1}_{\hat{P}^{k+1}}(\emptyset)|\mathbf{S}_{k+1} \subseteq T^{i+1}_{P^{k+1}}(\mathbf{I}_k)|\mathbf{S}_{k+1}.$$

(Here and later, $\subseteq$ denotes the usual 2-valued containment between instances; this is well defined because all instances considered are total, even if $\mathbf{J}$ is not.) In (1), the $3\text{-}T^{i+1}_{\hat{P}^{k+1}}$ and $T^{i+1}_{P^{k+1}}$ terms may not be equal, because the positive atoms of $\mathbf{I}_k = \mathbf{J}_k$ are available when applying $T_{P^{k+1}}$ the first time but are available only during the second application of $3\text{-}T_{\hat{P}^{k+1}}$. On the other hand, the $T^i_{P^{k+1}}$ and $3\text{-}T^{i+1}_{\hat{P}^{k+1}}$ terms may not be equal (e.g., if there is a rule of the form $A \leftarrow$ in P^{k+1}).

By (1) and finiteness of the input, there is some $m \geq 0$ such that for each $i \geq 0$,

(2) $$\mathbf{I}_n|\mathbf{S}_{k+1} = T^{m+i}_{P^{k+1}}(\mathbf{I}_k)|\mathbf{S}_{k+1} = 3\text{-}T^{m+i}_{\hat{P}^{k+1}}(\emptyset)|\mathbf{S}_{k+1}.$$

This is almost what is needed to complete the induction, except that $\hat{P}^{k+1}$ is used instead of $\hat{P}$. However, observe that for each $i \geq 0$,

(3) $$3\text{-}T^i_{\hat{P}}(\emptyset)|\mathbf{S}_{k+1} \subseteq 3\text{-}T^i_{\hat{P}^{k+1}}(\emptyset)|\mathbf{S}_{k+1}$$

because $3\text{-}T^i_{\hat{P}}(\emptyset)|\mathbf{S}_{[0,k]} \subseteq \mathbf{J}_k$ for each $i \geq 0$ by the induction hypothesis. Finally observe that for each $i \geq 0$,

(4) $$3\text{-}T^i_{\hat{P}^{k+1}}(\emptyset)|\mathbf{S}_{k+1} \subseteq 3\text{-}T^{i+l_k}_{\hat{P}}(\emptyset)|\mathbf{S}_{k+1}$$

because $3\text{-}T_{\hat{P}}^{l_k}(\emptyset)|\mathbf{S}_{[0,k]}$ contains all of the positive atoms of $\mathbf{J}_k$.

Then for each $i \geq 0$ we have

$$\begin{aligned} 3\text{-}T_{\hat{P}^{k+1}}^{m+i}(\emptyset)|\mathbf{S}_{k+1} &\subseteq 3\text{-}T_{\hat{P}}^{m+i+l_k}(\emptyset)|\mathbf{S}_{k+1} && \text{by (4)} \\ &\subseteq 3\text{-}T_{\hat{P}^{k+1}}^{m+i+l_k}(\emptyset)|\mathbf{S}_{k+1} && \text{by (3)} \\ &\subseteq 3\text{-}T_{\hat{P}^{k+1}}^{m+i}(\emptyset)|\mathbf{S}_{k+1} && \text{by (2).} \end{aligned}$$

It follows that

$$3\text{-}T_{\hat{P}^{k+1}}^{m+i}(\emptyset)|\mathbf{S}_{k+1} = 3\text{-}T_{\hat{P}}^{m+i+l_k}(\emptyset)|\mathbf{S}_{k+1}. \tag{5}$$

Set $l_{(k+1)} = l_k + m$. Combining (2) and (5), we have, for each $i \geq 0$,

$$\mathbf{J}|\mathbf{S}_{k+1} = 3\text{-}T_{\hat{P}}^{l_{(k+1)}+i}(\emptyset)|\mathbf{S}_{k+1} = \mathbf{I}_n|\mathbf{S}_{k+1}.$$

Together with the inductive hypothesis, we obtain for each $i \geq 0$ that

$$\mathbf{J}|\mathbf{S}_{[0,k+1]} = 3\text{-}T_{\hat{P}}^{l_{(k+1)}+i}(\emptyset)|\mathbf{S}_{[0,k+1]} = \mathbf{I}_n|\mathbf{S}_{[0,k+1]},$$

which concludes the proof. ■

As just seen, each stratifiable program is total under the well-founded semantics. However, as indicated by Example 15.3.8(b), a datalog$^\neg$ program P may yield a 3-valued model $P^{wf}(\mathbf{I})$ on some inputs. Furthermore, there are programs that are not stratified but whose well-founded models are nonetheless total (see Exercise 15.22). Unfortunately, there can be no effective characterization of those datalog$^\neg$ programs whose well-founded semantics is total for all input databases (Exercise 15.23). One can find sufficient syntactic conditions that guarantee the totality of the well-founded semantics, but this quickly becomes a tedious endeavor. It has been shown, however, that for each datalog$^\neg$ program P, one can find another program whose well-founded semantics is total on all inputs and that produces the same positive facts as the well-founded semantics of P.

15.4 Expressive Power

In this section, we examine the expressive power of datalog$^\neg$ with the various semantics for negation we have considered. More precisely, we focus on semipositive, stratified, and well-founded semantics. We first look at the relative power of these semantics and show that semipositive programs are weaker than stratified, which in turn are weaker than well founded. Then we look at the connection with languages studied in Chapter 14 that also use recursion and negation. We prove that well-founded semantics can express precisely the *fixpoint* queries.

Finally we look at the impact of *order* on expressive power. An ordered database contains a special binary relation *succ* that provides a successor relation on all constants

in the active domain. Thus the constants are ordered by *succ* and in fact can be viewed as integers. The impact of assuming that a database is ordered is examined at length in Chapter 17. Rather surprisingly, we show that in the presence of order, semipositive programs are as powerful as programs with well-founded semantics. In particular, all three semantics are equivalent and express precisely the *fixpoint* queries.

We begin by briefly noting the connection between stratified datalog$^\neg$ and relational calculus (and algebra). To see that stratified datalog$^\neg$ can express all queries in CALC, recall the nonrecursive datalog$^\neg$ (nr-datalog$^\neg$) programs introduced in Chapter 5. Clearly, these are stratified datalog$^\neg$ programs in which recursion is not allowed. Theorem 5.3.10 states that nr-datalog$^\neg$ (with one answer relation) and CALC are equivalent. It follows that stratified datalog$^\neg$ can express all of CALC. Because transitive closure of a graph can be expressed in stratified datalog$^\neg$ but not in CALC (see Proposition 17.2.3), it follows that stratified datalog$^\neg$ is strictly stronger than CALC.

Stratified Datalog Is Weaker than *Fixpoint*

Let us look at the expressive power of stratified datalog$^\neg$. Computationally, stratified programs provide recursion and negation and are inflationary. Therefore one might expect that they express the *fixpoint* queries. It is easy to see that all stratified datalog$^\neg$ are *fixpoint* queries (Exercise 15.28). In particular, this shows that such programs can be evaluated in polynomial time. Can stratified datalog$^\neg$ express all *fixpoint* queries? Unfortunately, no. The intuitive reason is that in stratified datalog$^\neg$ there is no recursion through negation, so the number of applications of negation is bounded. In contrast, *fixpoint* queries allow recursion through negation, so there is no bound on the number of applications of negation. This distinction turns out to be crucial. We next outline the main points of the argument, showing that stratified datalog$^\neg$ is indeed strictly weaker than *fixpoint*.

The proof uses a game played on so-called game trees. The game is played on a given tree. The nodes of the tree are the possible positions in the game, and the edges are the possible moves from one position to another. Additionally, some leaves of the tree are labeled black. The game is between two players. A round of the game starting at node x begins with Player I making a move from x to one of its children y. Player II then makes a move from y, etc. The game ends when a leaf is reached. Player I wins if Player II picks a black leaf. For a given tree (with labels), Player I has a winning strategy for the game starting at node x if he or she can win starting at x no matter how Player II plays. We are interested in programs determining whether there is such a winning strategy.

The game tree is represented as follows. The set of possible moves is given by a binary relation *move* and the set of black nodes by a unary relation *black*. Consider the query *winning* (not to be confused with the predicate *win* of Example 15.3.1), which asks if Player I has a winning strategy starting at the root of the tree. We will define a set of game trees $\mathcal{G}$ such that

(i) the query *winning* on the game trees in $\mathcal{G}$ is definable by a *fixpoint* query, and

(ii) for each stratified program P, there exist game trees $G, G' \in \mathcal{G}$ such that *winning* is true on G and false on G', but P cannot distinguish between G and G'.

Clearly, (ii) shows that the *winning* query on game trees is not definable by a stratified

datalog$^\neg$ program. The set $\mathcal{G}$ of game trees is defined next. It consists of the $G_{l,k}$ and $G'_{l,k}$ defined by induction as follows:

- $G_{0,k}$ and $G'_{0,k}$ have no moves and just one node, labeled black in $G_{0,k}$ and not labeled in $G'_{0,k}$.
- $G_{i+1,k}$ consists of a copy of $G'_{i,k}$, k disjoint copies of $G_{i,k}$, and a new root d_{i+1}. The moves are the union of the moves in the copies of $G'_{i,k}$ and $G_{i,k}$ together with new moves from the root d_{i+1} to the roots of the copies. The labels remain unchanged.
- $G'_{i+1,k}$ consists of $k+1$ disjoint copies of $G_{i,k}$ and a new root d'_{i+1} from which moves are possible to the roots of the copies of $G_{i,k}$.

The game trees $G_{4,1}$ and $G'_{4,1}$ are represented in Fig. 15.2. It is easy to see that *winning* is true on the game trees $G_{2i,k}$ and false on game trees $G'_{2i,k}$, $i > 0$ (Exercise 15.30).

We first note that the query *winning* on game trees in $\mathcal{G}$ can be defined by a *fixpoint* query. Consider

$$\begin{aligned}\varphi(T) = {} & (\exists y)[Move(x, y) \wedge (\forall z)(Move(y, z) \rightarrow Black(z))] \\ & \vee (\exists y)[Move(x, y) \wedge (\forall z)(Move(y, z) \rightarrow T(z))].\end{aligned}$$

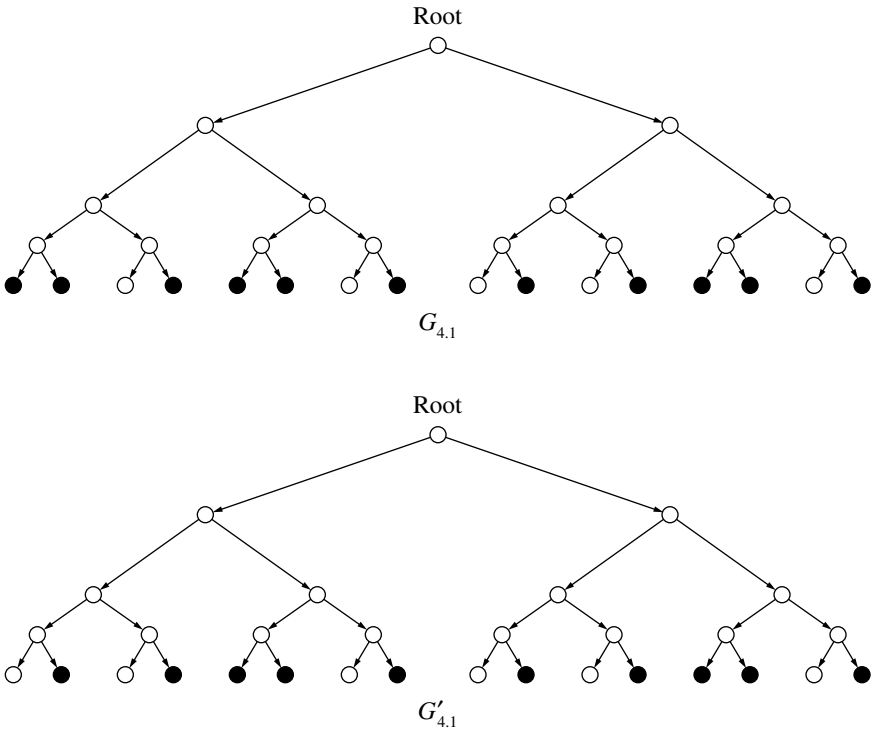


Figure 15.2: Game trees

It is easy to verify that *winning* is defined by $\mu_T(\varphi(T))(root)$, where *root* is the root of the game tree (Exercise 15.30). Next we note that the *winning* query is not expressible by any stratified datalog$^\neg$ program. To this end, we use the following result, stated without proof.

LEMMA 15.4.1 For each stratified datalog$^\neg$ program P, there exist i, k such that

$$P(G_{i,k})(winning) = P(G'_{i,k})(winning).$$

The proof of Lemma 15.4.1 uses an extension of Ehrefeucht-Fraissé games (the games are described in Chapter 17). The intuition of the lemma is that, to distinguish between $G_{i,k}$ and $G'_{i,k}$ for i and k sufficiently large, one needs to apply more negations than the fixed number allowed by P. Thus no stratified program can distinguish between all the $G_{i,k}$ and $G'_{i,k}$. In particular, it follows that the *fixpoint* query *winning* is not equivalent to *any* stratified datalog$^\neg$ program. Thus we have the following result, settling the relationship between stratified datalog$^\neg$ and the *fixpoint* queries.

THEOREM 15.4.2 The class of queries expressible by stratified datalog$^\neg$ programs is strictly included in the *fixpoint* queries.

REMARK 15.4.3 The game tree technique can also be used to prove that the number of strata in stratified datalog$^\neg$ programs has an impact on expressive power. Specifically, let $Strat_i$ consist of all queries expressible by stratified datalog$^\neg$ programs with i strata. Then it can be shown that for all i, $Strat_i \subset Strat_{i+1}$. In particular, semipositive datalog$^\neg$ is weaker than stratified datalog$^\neg$.

Well-Founded Datalog$^\neg$ Is Equivalent to *Fixpoint*

Next we consider the expressive power of datalog$^\neg$ programs with well-founded semantics. We prove that well-founded semantics can express precisely the *fixpoint* queries. We begin by showing that the well-founded semantics can be computed by a *fixpoint* query. More precisely, we show how to compute the set of false, true, and undefined facts of the answer using a *while*$^+$ program (see Chapter 14 for the definition of *while*$^+$ programs).

THEOREM 15.4.4 Let P be a datalog$^\neg$ program. There exists a *while*$^+$ program w with input relations $edb(P)$, such that

1. w contains, for each relation R in $sch(P)$, three relation variables R^ϵ_{answer}, where $\epsilon \in \{0, 1/2, 1\}$;
2. for each instance $\mathbf{I}$ over $edb(P)$, $u \in w(\mathbf{I})(R^\epsilon_{answer})$ iff $P^{wf}(\mathbf{I})(R(u)) = \epsilon$, for $\epsilon \in \{0, 1/2, 1\}$.

Crux Let P be a datalog$^\neg$ program. The *while*$^+$ program mimics the alternating fixpoint computation of P^{wf}. Recall that this involves repeated applications of the operator $conseq_P$, resulting in the sequence

$$\mathbf{I}_0 \prec \mathbf{I}_2 \ldots \prec \mathbf{I}_{2i} \prec \mathbf{I}_{2i+2} \prec \ldots \prec \mathbf{I}_{2i+1} \prec \mathbf{I}_{2i-1} \prec \ldots \prec \mathbf{I}_1.$$

Recall that the $\mathbf{I}_i$ are all total instances. Thus 3-valued instances are only required to produce the final answer from $\mathbf{I}_*$ and $\mathbf{I}^*$ at the end of the computation, by one last first-order query.

It is easily verified that *while*$^+$ can simulate one application of $conseq_P$ on total instances (Exercise 15.27). The only delicate point is to make sure the computation is inflationary. To this end, the program w will distinguish between results of even and odd iterations of $conseq_P$ by having, for each R, an odd and even version R^0_{odd} and R^1_{even}. R^0_{odd} holds at iteration $2i+1$ the negative facts of R in $\mathbf{I}_{2i+1}$, and R^1_{even} holds at iteration $2i$ the positive facts of R in $\mathbf{I}_{2i}$. Note that both R^0_{odd} and R^1_{even} are increasing throughout the computation.

We elaborate on the simulation of the operator $conseq_P$ on a total instance $\mathbf{I}$. The program w will have to distinguish between facts in the input $\mathbf{I}$, used to resolve the negative premises of rules in P, and those inferred by applications of 3-T_P. Therefore for each relation R, the *while*$^+$ program will also maintain a copy $\bar{R}_{even}$ and $\bar{R}_{odd}$ to hold the facts produced by consecutive applications of 3-T_P in the even and odd cases, respectively. More precisely, the $\bar{R}_{odd}$ hold the positive facts inferred from input $\mathbf{I}_{2i}$ represented in R^1_{even}, and the $\bar{R}_{even}$ hold the positive facts inferred from input $\mathbf{I}_{2i+1}$ represented in R^0_{odd}. It is easy to write a first-order query defining one application of 3-T_P for the even or odd cases. Because the representations of the input are different in the even and odd cases, different programs must be used in the two cases. This can be iterated in an inflationary manner, because the set of positive facts inferred in consecutive applications of 3-T_P is always increasing. However, the $\bar{R}_{odd}$ and $\bar{R}_{even}$ have to be initialized to $\emptyset$ at each application of $conseq_P$. Because the computation must be inflationary, this cannot be done directly. Instead, timestamping must be used. The initialization of the $\bar{R}_{odd}$ and $\bar{R}_{even}$ is simulated by timestamping each relation with the current content of R^1_{even} and R^0_{odd}, respectively. This is done in a manner similar to the proofs of Chapter 14. ■

We now exhibit a converse of Theorem 15.4.4, showing that any *fixpoint* query can essentially be simulated by a datalog$^\neg$ program with well-founded semantics. More precisely, the positive portion of the well-founded semantics yields the same facts as the *fixpoint* query.

Example 15.4.6 illustrates the proof of this result.

THEOREM 15.4.5 Let q be a *fixpoint* query over input schema $\mathbf{R}$. There exists a datalog$^\neg$ program P such that $edb(P) = \mathbf{R}$, P has an *idb* relation *answer*, and for each instance $\mathbf{I}$ over $\mathbf{R}$, the positive portion of *answer* in $P^{wf}(\mathbf{I})$ coincides with $q(\mathbf{I})$.

Crux We will use the definition of *fixpoint* queries by iterations of positive first-order formulas. Let q be a *fixpoint* query. As discussed in Chapter 14, there exists a CALC formula $\varphi(T)$, positive in T, such that q is defined by $\mu_T(\varphi(T))(u)$, where u is a vector of variables and constants. Consider the CALC formula $\varphi(T)$. As noted earlier in this section, there is an nr-datalog$^\neg$ program P_φ with one answer relation R' such that P_φ is equivalent

to $\varphi(T)$. Because $\varphi(T)$ is positive in T, along any path in the syntax tree of $\varphi(T)$ ending with atom T there is an even number of negations. This is also true of paths in G_{P_φ}.

Consider the precedence graph G_{P_φ} of P_φ. Clearly, one can construct P_φ such that each *idb* relation except T is used in the definition of exactly one other *idb* relation, and all idb relations are used eventually in the definition of the answer R'. In other words, for each *idb* relation R other than T, there is a unique path in G_{P_φ} from R to R'. Consider the paths from T to some *idb* relation R in P_φ. Without loss of generality, we can assume that all paths have the same number of negations (otherwise, because all paths to T have an even number of negations, additional *idb* relations can be introduced to pad the paths with fewer negations, using rules that perform redundant double negations). Let the *rank* of an *idb* relation R in P_φ be the number of negations on each path leading from T to R in G_{P_φ}. Now let P be the datalog$^\neg$ program obtained from P_φ as follows:

- replace the answer relation R' by T;
- add one rule $answer(v) \leftarrow T(u)$, where v is the vector of distinct variables occurring in u, in order of occurrence.

The purpose of replacing R' by T is to cause program P_φ to iterate, yielding $\mu_T(\varphi(T))$. The last rule is added to perform the final selection and projection needed to obtain the answer $\mu_T(\varphi(T))(u)$. Note that, in some sense, P is almost stratified, except for the fact that the result T is fed back into the program.

Consider the alternating fixpoint sequence $\{\mathbf{I}_i\}_{i\geq 0}$ in the computation of $P^{wf}(\mathbf{I})$. Suppose R' has rank q in P_φ, and let R be an *idb* relation of P_φ whose rank in P_φ is $r \leq q$. Intuitively, there is a close correspondence between the sequence $\{\mathbf{I}_i\}_{i\geq 0}$ and the iterations of φ, along the following lines: Each application of $conseq_P$ propagates the correct result from relations of rank r in P_φ to relations of rank $r+1$. There is one minor glitch, however: In the fixpoint computation, the *edb* relations are given, and even at the first iteration, their negation is taken to be their complement; in the alternating fixpoint computation, all negative literals, including those involving *edb* relations, are initially taken to be true. This results in a mismatch. To fix the problem, consider a variation of the alternating fixpoint computation of $P^{wf}(\mathbf{I})$ defined as follows:

$$\begin{aligned}\bar{\mathbf{I}}_0 &= \mathbf{I} \cup \neg.\{R(a_1,\ldots,a_n) \mid R \in idb(P),\ R(a_1,\ldots,a_n) \in \mathbf{B}(P,\mathbf{I})\}\\ \bar{\mathbf{I}}_{i+1} &= conseq_P(\bar{\mathbf{I}}_i).\end{aligned}$$

Clearly, $\perp \prec \bar{\mathbf{I}}_0 \prec P^{wf}(\mathbf{I})$. Then, by Proposition 15.3.10, $\bar{\mathbf{I}}^*_* = P^{wf}(\mathbf{I})$.

Now the following can be verified by induction for each *idb* relation R of rank r:

For each i, $(\bar{\mathbf{I}}_{iq+r})^1$ contains exactly the facts of R true in $P_\varphi(\varphi^i(\emptyset))$.

Intuitively, this is so because each application of $conseq_P$ propagates the correct result across one application of negation to an *idb* predicate. Because R' has rank q, it takes q applications to simulate a complete application of P_φ. In particular, it follows that for each i, $(\bar{\mathbf{I}}_{iq})^1$ contains in T the facts true in $\varphi^i(\emptyset)$.

Thus $(\bar{\mathbf{I}}_*)^1$ contains in T the facts true in $\mu_T(\varphi(T))$. Finally *answer* is obtained by a simple selection and projection from T using the last rule in P and yields $\mu_T(\varphi(T))(u)$. ■

In the preceding theorem, the *positive* portion of *answer* for $P^{wf}(\mathbf{I})$ coincides with $q(\mathbf{I})$. However, $P^{wf}(\mathbf{I})$ is not guaranteed to be total (i.e., it may contain unknown facts). Using a recent result (not demonstrated here), a program Q can be found such that Q^{wf} always provides a total answer, and such that the positive facts of P^{wf} and Q^{wf} coincide on all inputs.

Recall from Chapter 14 that datalog$^\neg$ with inflationary semantics also expresses precisely the *fixpoint* queries. Thus we have converged again, this time by the deductive database path, to the *fixpoint* queries. This bears witness, once more, to the naturalness of this class. In particular, the well-founded and inflationary semantics, although very different, have the same expressive power (modulo the difference between 3-valued and 2-valued models).

Example 15.4.6 Consider the *fixpoint* query $\mu_{good}(\varphi(good))(x)$, where

$$\varphi(good) = \forall y(G(y, x) \rightarrow good(y)).$$

Recall that this query, also encountered in Chapter 14, computes the "good" nodes of the graph G (i.e., those that cannot be reached from a cycle). The nr-datalog$^\neg$ program P_φ corresponding to one application of $\varphi(good)$ is the one exhibited in Example 15.3.8(c):

$$\begin{aligned} bad(x) &\leftarrow G(y, x), \neg good(y) \\ R'(x) &\leftarrow \neg bad(x) \end{aligned}$$

Note that *bad* is negative in P_φ and has rank one, and *good* is positive. The answer R' has rank two. The program P is as follows:

$$\begin{aligned} bad(x) &\leftarrow G(y, x), \neg good(y) \\ good(x) &\leftarrow \neg bad(x) \\ answer(x) &\leftarrow good(x) \end{aligned}$$

Consider the input graph

$$G = \{\langle b, c\rangle, \langle c, b\rangle, \langle c, d\rangle, \langle a, d\rangle, \langle a, e\rangle\}.$$

The consecutive values of $\varphi^i(\emptyset)$ are

$$\begin{aligned} \varphi(\emptyset) &= \{a\}, \\ \varphi^2(\emptyset) &= \{a, e\}, \\ \varphi^3(\emptyset) &= \{a, e\}. \end{aligned}$$

Thus $\mu_{good}(\varphi(good))(x)$ yields the answer $\{a, e\}$. Consider now the alternating fixpoint sequence in the computation of P^{wf} on the same input (only the positive facts of *bad* and *good* are listed, because G does not change and $answer = good$).

	bad	*good*
$\bar{\mathbf{I}}_0$	$\emptyset$	$\emptyset$
$\bar{\mathbf{I}}_1$	$\{b, c, d, e\}$	$\{a, b, c, d, e\}$
$\bar{\mathbf{I}}_2$	$\emptyset$	$\{a\}$
$\bar{\mathbf{I}}_3$	$\{b, c, d\}$	$\{a, b, c, d, e\}$
$\bar{\mathbf{I}}_4$	$\emptyset$	$\{a, e\}$
$\bar{\mathbf{I}}_5$	$\{b, c, d\}$	$\{a, b, c, d, e\}$
$\bar{\mathbf{I}}_6$	$\emptyset$	$\{a, e\}$

Thus

$$\varphi(\emptyset) = (\bar{\mathbf{I}}_2)^1(good),$$
$$\varphi^2(\emptyset) = (\bar{\mathbf{I}}_4)^1(good)$$

and

$$(\bar{\mathbf{I}}_4)^1(answer) = \mu_{good}(\varphi(good))(x).$$

The relative expressive power of the various languages discussed in this chapter is summarized in Fig. 15.3. The arrows indicate strict inclusion. For a view of these languages in a larger context, see also Figs. 18.4 and 18.5 at the end of Part E.

The Impact of Order

Finally we look at the impact of order on the expressive power of the various datalog$^\neg$ semantics. As we will discuss at length in Chapter 17, the assumption that databases are ordered can have a dramatic impact on the expressive power of languages like *fixpoint* or *while*. The datalog$^\neg$ languages are no exception. The effect of order is spectacular. With this assumption, it turns out that semipositive datalog$^\neg$ is (almost) as powerful as stratified datalog$^\neg$ and datalog$^\neg$ with well-founded semantics. The "almost" comes from a

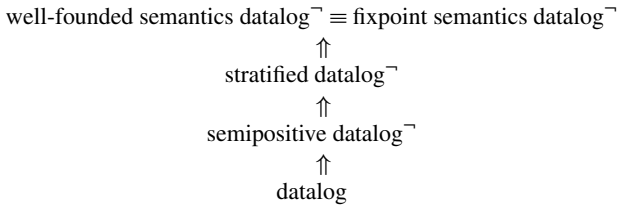


Figure 15.3: Relative expressive power of datalog$^{(\neg)}$ languages

technicality concerning the order: We also need to assume that the minimum and maximum constants are explicitly given. Surprisingly, these constants, which can be computed with a first order query if *succ* is given, cannot be computed with semipositive programs (see Exercise 15.29).

The next lemma states that semipositive programs express the *fixpoint* queries on ordered databases with *min* and *max* (i.e., databases with a predicate *succ* providing a successor relation among all constants, and unary relations *min* and *max* containing the smallest and the largest constant).

Lemma 15.4.7 The semipositive datalog$^\neg$ programs express precisely the *fixpoint* queries on ordered databases with *min* and *max*.

Crux Let q be a *fixpoint* query over database schema $\mathbf{R}$. Because q is a *fixpoint* query, there is a first-order formula $\varphi(T)$, positive in T, such that q is defined by $\mu_T(\varphi(T))(u)$, where u is a vector of variables and constants. Because T is positive in $\varphi(T)$, we can assume that $\varphi(T)$ is in prenex normal form $Q_1x_1Q_2x_2\ldots Q_kx_k(\psi)$, where ψ is a quantifier free formula in disjunctive normal form and T is not negated in ψ. We show by induction on k that there exists a semipositive datalog$^\neg$ program P_φ with an *idb* relation $answer_\varphi$ defining $\mu_T(\varphi(T))$ [the last selection and projection needed to obtain the final answer $\mu_T(\varphi(T))(u)$ pose no problem]. Suppose $k = 0$ (i.e., $\varphi = \psi$). Then P_φ is the nr-datalog$^\neg$ program corresponding to ψ, where the answer relation is T. Because ψ is quantifier free and T is not negated in ψ, P_φ is clearly semipositive. Next suppose the statement is true for some $k \geq 0$, and let $\varphi(T)$ have quantifier depth $k+1$. There are two cases:

(i) $\varphi = \exists x\psi(x, v)$, where ψ has quantifier depth k. Then P_φ contains the rules of P_ψ, where T is replaced in heads of rules by a new predicate T' and one additional rule

$$T(v) \leftarrow T'(x, v).$$

(ii) $\varphi = \forall x\psi(x, v)$, where ψ has quantifier depth k. Then P_φ consists, again, of P_ψ, where T is replaced in heads of rules by a new predicate T', with the following rules added:

$$\begin{aligned} R'(x, v) &\leftarrow T'(x, v), min(x) \\ R'(x', v) &\leftarrow R'(x, v), succ(x, x'), T'(x', v) \\ T(v) &\leftarrow R'(x, v), max(x), \end{aligned}$$

where R' is a new auxiliary predicate. Thus the program steps through all x's using the successor relation *succ*, starting from the minimum constant. If the maximum constant is reached, then $T'(x, v)$ is satisfied for all x, and $T(v)$ is inferred.

This completes the induction. ■

As we shall see in Chapter 17, *fixpoint* expresses on ordered databases exactly the

queries computable in time polynomial in the size of the database (i.e., QPTIME). Thus we obtain the following result. In comparing well-founded semantics with the others, we take the positive portion of the well-founded semantics as the answer.

THEOREM 15.4.8 Stratified datalog$^\neg$ and datalog$^\neg$ with well-founded semantics are equivalent on ordered databases and express exactly QPTIME. They are also equivalent to semipositive datalog$^\neg$ on ordered databases with *min* and *max* and express exactly QPTIME.

15.5 Negation as Failure in Brief

In our presentation of datalog in Chapter 12, we saw that the minimal model and least fixpoint semantics have an elegant proof-theoretic counterpart based on SLD resolution. One might naturally wonder if such a counterpart exists in the case of datalog$^\neg$. The answer is yes and no. Such a proof-theoretic approach has indeed been proposed and is called negation as failure. This was originally developed for logic programming and predates stratified and well-founded semantics. Unfortunately, the approach has two major drawbacks. The first is that it results in a proof-building procedure that does not always terminate. The second is that it is not the exact counterpart of any other existing semantics. The semantics that has been proposed as a possible match is "Clark's completion," but the match is not perfect and Clark's completion has its own problems. We provide here only a brief and informal presentation of negation as failure and the related Clark's completion.

The idea behind negation as failure is simple. We would like to infer a negative fact $\neg A$ if A cannot be proven by SLD resolution. Thus $\neg A$ would then be proven by the failure to prove A. Unfortunately, this is generally noneffective because SLD derivations may be arbitrarily long, and so one cannot check in finite time[2] that there is no proof of A by SLD resolution. Instead we have to use a weaker notion of negation by failure, which can be checked. This is done as follows. A fact $\neg A$ is proven if all SLD derivations starting from the goal $\leftarrow A$ are finite and none produces an SLD refutation for $\leftarrow A$. In other words, A *finitely fails*. This procedure applies to ground atoms A only. It gives rise to a proof procedure called SLDNF resolution. Briefly, SLDNF resolution extends SLD resolution as follows. Refutations of positive facts proceed as for SLD resolution. Whenever a negative ground goal $\leftarrow \neg A$ has to be proven, SLD resolution is applied to $\leftarrow A$, and $\neg A$ is proven if the SLD resolution finitely fails for $\leftarrow A$. The idea of SLDNF seems appealing as the proof-theoretic version of the closed world assumption. However, as illustrated next, it quickly leads to significant problems.

EXAMPLE 15.5.1 Consider the usual program P_{TC} for transitive closure of a graph:

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

---

[2] Because databases are finite, one can develop mechanisms to bound the expansion. We ignore this aspect here.

Consider the instance **I** where G has edges $\{\langle a, b\rangle, \langle b, a\rangle, \langle c, a\rangle\}$. Clearly, $\{\langle a, c\rangle\}$ is not in the transitive closure of G, and so not in T, by the usual datalog semantics. Suppose we wish to prove the fact $\neg T(a, c)$, using negation as failure. We have to show that SLD resolution finitely fails on $T(a, c)$, with the preceding program and input. Unfortunately, SLD resolution can enter a negative loop when applied to $\leftarrow T(a, c)$. One obtains the following SLD derivation:

1. $\leftarrow T(a, c)$;
2. $\leftarrow G(a, z), T(z, c)$, using the second rule;
3. $\leftarrow T(b, c)$, using the fact $G(a, b)$;
4. $\leftarrow G(b, z), T(z, c)$ using the second rule;
5. $\leftarrow T(a, c)$ using the fact $G(b, a)$.

Note that the last goal is the same as the first, so this can be extended to an infinite derivation. It follows that SLD resolution does not finitely fail on $\leftarrow T(a, c)$, so SLDNF does not yield a proof of $\neg T(a, c)$. Moreover, it has been shown that this does not depend on the particular program used to define transitive closure. In other words, there is *no* datalog$^{\neg}$ program that under SLDNF can prove the positive and negative facts true of the transitive closure of a graph.

The preceding example shows that SLDNF can behave counterintuitively, even in some simple cases. The behavior is also incompatible with all the semantics for negation that we have discussed so far. Thus one cannot hope for a match between SLDNF and these semantics.

Instead a semantics called Clark's completion has been proposed as a candidate match for negation as failure. It works as follows. For a datalog$^{\neg}$ program P, the *completion of* P, *comp(P)*, is constructed as follows. For each *idb* predicate R, each rule

$$\rho : R(u) \leftarrow L_1(v_1), \ldots, L_n(v_n)$$

defining R is rewritten so there is a uniform set of distinct variables in the rule head and so all free variables in the body are existentially quantified:

$$\rho' : R(u') \leftarrow \exists v'(x_1 = t_1 \wedge \cdots \wedge x_k = t_k \wedge L_1(v_1) \wedge \cdots \wedge L_n(v_n)).$$

(If the head of ρ has distinct variables for all coordinates, then the equality atoms can be avoided. If repeated variables or constants occur, then equality must be used.) Next, if the rewritten rules for R are $\rho'_1, \ldots, \rho'_l$, the *completion* of R is formed by

$$\forall u'(R(u') \leftrightarrow body(\rho'_1) \vee \cdots \vee body(\rho'_l)).$$

Intuitively, this states that ground atom $R(w)$ is true iff it is supported by one of the rules defining R. Finally the completion of P is the set of completions of all *idb* predicates of P, along with the axioms of equality, if needed.

The semantics of P is now defined by the following: A is true iff it is a logical consequence of $comp(P)$. A first problem now is that $comp(P)$ is not always consistent; in fact, its consistency is undecidable. What is the connection between SLDNF and Clark's completion? Because SLDNF is consistent (it clearly cannot prove A and $\neg A$) and $comp(P)$ is not so always, SLDNF is not always complete with respect to $comp(P)$. For consistent $comp(P)$, it can be shown that SLDNF resolution is sound. However, additional conditions must be imposed on the datalog$^\neg$ programs for SLDNF resolution to be complete.

Consider again the transitive closure program P_{TC} and input instance $\mathbf{I}$ of Example 15.5.1. Then the completion of T is equivalent to

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

Note that neither $T(a, c)$ nor $\neg T(a, c)$ are consequences of $comp(P_{TC,\mathbf{I}})$.

In summary, negation as failure does not appear to provide a convincing proof-theoretic counterpart to the semantics we have considered. The search for more successful proof-theoretic approaches is an active research area. Other proposals are described briefly in the Bibliographic Notes.

Bibliographic Notes

The notion of a stratified program is extremely natural. Not surprisingly, it was proposed independently by quite a few investigators [CH85, ABW88, Lif88, VanG86]. The independence of the semantics from a particular stratification (Theorem 15.2.10) was shown in [ABW88].

Research on well-founded semantics, and the related notion of a 3-stable model, has its roots in investigations of stable and default model semantics. Although formulated somewhat differently, the notion of a stable/default model is equivalent to that of a total 3-stable model [Prz90]. Stable model semantics was introduced in [GL88], and default model semantics was introduced in [BF87, BF88]. Stable semantics is based on Moore's autoepistemic logic [Moo85], and default semantics is based on Reiter's default logic [Rei80]. The equivalence between autoepistemic and default logic in the general case has been shown in [Kon88]. The equivalence between stable model semantics and default model semantics was shown in [BF88].

Several equivalent definitions of the well-founded semantics have been proposed. The definition used in this chapter comes from [Prz90]. The alternating fixpoint computation we described is essentially the same as in [VanG89]. Alternative procedures for computing the well-founded semantics are exhibited in [BF88, Prz89]. Historically, the first definition of well-founded semantics was proposed in [VanGRS88, VanGRS91]. This is described in Exercise 15.24.

The fact that well-founded and stratified semantics agree on stratifiable datalog$^\neg$ programs (Theorem 15.3.11) was shown in [VanGRS88].

Both the stratified and well-founded semantics were originally introduced for general logic programming, as well as the more restricted case of datalog. In the context of logic programming, both semantics have expressive power equivalent to the arithmetic hierarchy [AW88] and are thus noneffective.

The result that datalog$^\neg$ with well-founded semantics expresses exactly the *fixpoint*

queries is shown in [VanG89]. Citation [FKL87] proves that for every datalog¬ program P there is a *total* datalog¬ program Q such that the positive portions of $P^{wf}(\mathbf{I})$ and $Q^{wf}(\mathbf{I})$ coincide for every $\mathbf{I}$. The fact that stratified datalog¬ is weaker than *fixpoint*, and therefore weaker than well-founded semantics, was shown in [Kol91], making use of earlier results from [Dal87] and [CH82]. In particular, Lemma 15.4.1 is based on Lemma 3.9 in [CH82]. The result that semipositive datalog¬ expresses QPTIME on ordered databases with *min* and *max* is due to [Pap85].

The investigation of negation as failure was initiated in [Cla78], in connection with general logic programming. In particular, SLDNF resolution as well as Clark's completion are introduced there. The fact that there is no datalog¬ program for which the positive and negative facts about the transitive closure of the graph can be proven by SLDNF resolution was shown in [Kun88]. Other work related to Clark's completion can be found in [She88, Llo87, Fit85, Kun87].

Several variations of SLDNF resolutions have been proposed. SLS resolution is introduced in [Prz88] to deal with stratified programs. An exact match is achieved between stratified semantics and the proof procedure provided by SLS resolution. Although SLS resolution is effective in the context of (finite) databases, it is not so when applied to general logic programs, with function symbols. To deal with this shortcoming, several restrictions of SLS resolution have been proposed that are effective in the general framework [KT88, SI88].

Several proof-theoretic approaches corresponding to the well-founded semantics have been proposed. SLS resolution is extended from stratified to arbitrary datalog¬ programs in [Prz88], under well-founded semantics. Independently, another extension of SLS resolution called global SLS resolution is proposed in [Ros89], with similar results. These proposals yield noneffective resolution procedures. An effective procedure is described in [BL90].

In [SZ90], an interesting connection between nondeterminism and stable models of a program (i.e., total 3-stable models; see also Exercise 15.20) is pointed out. Essentially, it is shown that the stable models of a datalog¬ program can be viewed as the result of a natural nondeterministic choice. This uses the choice construct introduced earlier in [KN88]. Another use of nondeterminism is exhibited in [PY92], where an extension of well-founded semantics is provided, which involves the nondeterministic choice of a fixpoint of a datalog¬ program. This is called *tie-breaking* semantics. A discussion of nondeterminism in deductive databases is provided in [GPSZ91].

Another semantics in the spirit of well-founded is the valid model semantics introduced in [BRSS92]. It is less conservative than well-founded semantics, in the sense that all facts that are positive in well-founded semantics are also positive in the valid model semantics, but the latter generally yields more positive facts than well-founded semantics.

There are a few prototypes (but no commercial system) implementing stratified datalog¬. The language LDL [NT89, BNR+87, NK88] implements, besides the stratified semantics for datalog¬, an extension to complex objects (see also Chapter 20). The implementation uses heuristics based on the magic set technique described in Chapter 13. The language NAIL! (Not Yet Another Implementation of Logic!), developed at Stanford, is another implementation of the stratified semantics, allowing function symbols and a set construct. The implementation of NAIL! [MUG86, Mor88] uses a battery of evaluation techniques, including magic sets. The language EKS [VBKL89], developed at

ECRC (European Computer-Industry Research Center) in Munich, implements the stratified semantics and extensions allowing quantifiers in rule bodies, aggregate functions, and constraint specification. The CORAL system [RSS92, RSSS93] provides a database programming language that supports both imperative and deductive capabilities, including stratification. An implementation of well-founded semantics is described in [CW92].

Nicole Bidoit's survey on negation in databases [Bid91b], as well as her book on datalog [Bid91a], provided an invaluable source of information and inspired our presentation of the topic.

Exercises

Exercise 15.1

(a) Show that, for datalog$^\neg$ programs P, the immediate consequence operator T_P is not always monotonic.

(b) Exhibit a datalog$^\neg$ program P (using negation at least once) such that T_P *is* monotonic.

(c) Show that it is decidable, given a datalog$^\neg$ program P, whether T_P is monotonic.

Exercise 15.2 Consider the datalog$^\neg$ program $P_3 = \{p \leftarrow \neg r;\ r \leftarrow \neg p;\ p \leftarrow \neg p, r\}$. Verify that T_{P_3} has a least fixpoint, but T_{P_3} does not converge when starting on $\emptyset$.

Exercise 15.3

(a) Exhibit a datalog$^\neg$ program P and an instance $\mathbf{K}$ over $sch(P)$ such that $\mathbf{K}$ is a model of Σ_P but not a fixpoint of T_P.

(b) Show that, for datalog$^\neg$ programs P, a minimal fixpoint of T_P is not necessarily a minimal model of Σ_P and, conversely, a minimal model of Σ_P is not necessarily a minimal fixpoint of T_P.

Exercise 15.4 Prove Lemma 15.2.8.

Exercise 15.5 Consider a database for the Parisian metro and bus lines, consisting of two relations *Metro*[*Station, Next-Station*] and *Bus*[*Station, Next-Station*]. Write stratifiable datalog$^\neg$ programs to answer the following queries.

(a) Find the pairs of stations $\langle a, b\rangle$ such that one can go from a to b by metro but not by bus.

(b) A *pure bus path* from a to b is a bus itinerary from a to b such that for all consecutive stops c, d along the way, one cannot go from c to d by metro. Find the pairs of stations $\langle a, b\rangle$ such that there is a pure bus path from a to b.

(c) Find the pairs of stations $\langle a, b\rangle$ such that b can be reached from a by some combination of metro or bus, but not by metro or bus alone.

(d) Find the pairs of stations $\langle a, b\rangle$ such that b can be reached from a by some combination of metro or bus, but there is no pure bus path from a to b.

(e) The metro is useless in a bus path from a to b if by taking the metro at any intermediate point c one can return to c but not reach any other station along the path. Find the pairs of stations $\langle a, b\rangle$ such that the metro is useless in all bus paths connecting a and b.

Exercise 15.6 The semantics of stratifiable datalog¬ programs can be extended to infinite databases as follows. Let P be a stratifiable datalog¬ program and let $\sigma = P^1 \dots P^n$ be a stratification for P. For each (finite or infinite) instance $\mathbf{I}$ over $edb(P)$, $\sigma(\mathbf{I})$ is defined similarly to the finite case. More precisely, consider the sequence

$$\mathbf{I}_0 = \mathbf{I}$$
$$\mathbf{I}_i = P^i(\mathbf{I}_{i-1}|edb(P^i))$$

where

$$P^i(\mathbf{I}_{i-1}|edb(P^i)) = \cup_{j>0} T^j_{P_i}(\mathbf{I}_{i-1}|edb(P^i)).$$

Note that the definition is now noneffective because $P^i(\mathbf{I}_{i-1}|edb(P^i))$ may be infinite.

Consider a database consisting of one binary relation *succ* providing a successor relation on an infinite set of constants. Clearly, one can identify these constants with the positive integers.

(a) Write a stratifiable datalog¬ program defining a unary relation *prime* containing all constants in *succ* corresponding to primes.

(b) Write a stratifiable datalog¬ program P defining a 0-ary relation *Fermat*, which is true iff Fermat's Last Theorem[3] is true. (No shortcuts, please: The computation of the program should provide a proof of Fermat's Last Theorem, not just coincidence of truth value!)

Exercise 15.7 Prove Theorem 15.2.2.

Exercise 15.8 A datalog¬ program is *nonrecursive* if its precedence graph is acyclic. Show that every nonrecursive stratifiable datalog¬ program is equivalent to an nr-datalog¬ program, and conversely.

Exercise 15.9 Let $(A, <)$ be a partially ordered set. A listing $a_1, \dots, a_n$ of the elements in A is *compatible with* $<$ iff for $i < j$ it is not the case that $a_j < a_i$. Let σ', σ'' be listings of A compatible with $<$. Prove that one can obtain σ'' from σ' by a sequence of exchanges of adjacent elements a_l, a_m such that $a_l \not< a_m$ and $a_m \not< a_l$.

Exercise 15.10 Prove Lemma 15.2.9.

Exercise 15.11 (Supported models) Prove that there exist stratified datalog¬ programs P_1, P_2 such that $sch(P_1) = sch(P_2)$, $\Sigma_{P_1} \equiv \Sigma_{P_2}$, and there is a minimal model $\mathbf{I}$ of Σ_{P_1} such that $\mathbf{I}$ is a supported model for P_1, but not for P_2. (In other words, the notion of supported model depends not only on Σ_P, but also on the syntax of P.)

Exercise 15.12 Prove part (b) of Proposition 15.2.11.

Exercise 15.13 Prove Proposition 15.2.12.

♠ **Exercise 15.14** [Bid91b] (Local stratification) The following extension of the notion of stratification has been proposed for general logic programs [Prz86]. This exercise shows that local stratification is essentially the same as stratification for the datalog¬ programs considered in this chapter (i.e., without function symbols).

---

[3] Fermat's Last Theorem: There is no $n > 2$ such that the equation $a^n + b^n = c^n$ has a solution in the positive integers.

A datalog$^\neg$ program P is *locally stratified* iff for each $\mathbf{I}$ over $edb(P)$, $ground(P_\mathbf{I})$ is stratified. [An example of a locally stratified logic program with function symbols is $\{even(0) \leftarrow; even(s(x)) \leftarrow \neg even(x)\}$.] The semantics of a locally stratified program P on input $\mathbf{I}$ is the semantics of the stratified program $ground(P_\mathbf{I})$.

(a) Show that, if the rules of P contain no constants, then P is locally stratified iff it is stratified.

(b) Give an example of a datalog$^\neg$ program (with constants) that is locally stratified but not stratified.

(c) Prove that, for each locally stratified datalog$^\neg$ program P, there exists a stratified datalog$^\neg$ program equivalent to P.

Exercise 15.15 Let α and β be propositional Boolean formulas (using $\wedge, \vee, \neg, \rightarrow$). Prove the following:

(a) If α and β are equivalent with respect to 3-valued instances, then they are equivalent with respect to 2-valued instances.

(b) If α and β are equivalent with respect to 2-valued instances, they are not necessarily equivalent with respect to 3-valued instances.

Exercise 15.16 Prove Lemma 15.3.4.

Exercise 15.17 Let P be a datalog$^\neg$ program. Recall the definition of positivized ground version of P given $\mathbf{I}$, denoted $pg(P, \mathbf{I})$, where $\mathbf{I}$ is a 3-valued instance. Prove the following:

(a) If $\mathbf{I}$ is total, then $pg(P, \mathbf{I})$ is total.

(b) Let $\{\mathbf{I}_i\}_{i \geq 0}$ be the sequence of instances defined by

$$\begin{aligned} \mathbf{I}_0 &= \bot \\ \mathbf{I}_{i+1} &= pg(P, \mathbf{I}_i)(\bot) = conseq_P(\mathbf{I}_i). \end{aligned}$$

Prove that

$$\mathbf{I}_0 \prec \mathbf{I}_2 \cdots \prec \mathbf{I}_{2i} \prec \mathbf{I}_{2i+2} \prec \cdots \prec \mathbf{I}_{2i+1} \prec \mathbf{I}_{2i-1} \prec \cdots \prec \mathbf{I}_1.$$

Exercise 15.18 Exhibit a datalog$^\neg$ program that yields the complement of the transitive closure under well-founded semantics.

Exercise 15.19 Prove that for each datalog$^\neg$ program P and instance $\mathbf{I}$ over $edb(P)$, $P^{wf}(\mathbf{I})$ is a minimal 3-valued model of P whose restriction to $edb(P)$ equals $\mathbf{I}$.

♠ **Exercise 15.20** A total 3-stable model of a datalog$^\neg$ program P is called a *stable model of* P [GL88] (also called a *default model* [BF87, BF88]).

(a) Provide examples of datalog$^\neg$ programs that have (1) no stable models, (2) a unique stable model, and (3) several stable models.

(b) Show that P^{wf} is total iff all 3-stable models are total.

(c) Prove that, if P^{wf} is total, then P has a unique stable model, but the converse is false.

♠ **Exercise 15.21** [BF88] Let P be a datalog$^\neg$ program and $\mathbf{I}$ an instance over $edb(P)$. Prove that the problem of determining whether $P_\mathbf{I}$ has a stable model is NP-complete in the size of $P_\mathbf{I}$.

Exercise 15.22 Give an example of a datalog$^\neg$ program P such that P is not stratified but P^{wf} is total.

★ **Exercise 15.23** Prove that it is undecidable if the well-founded semantics of a given datalog$^\neg$ program P is always total. That is, it is undecidable whether, for each instance $\mathbf{I}$ over $edb(P)$, $P_{\mathbf{I}}^{wf}$ is total.

♠ **Exercise 15.24** [VanGRS88] This exercise provides an alternative (and historically first) definition of well-founded semantics. Let L be a ground literal. The *complement of* L is $\neg A$ if $L = A$ and A if $L = \neg A$. If $\mathbf{I}$ is a set of ground literals, we denote by $\neg.\mathbf{I}$ the set of complements of the literals in $\mathbf{I}$. A set $\mathbf{I}$ of ground literals is *consistent* iff $\mathbf{I} \cap \neg.\mathbf{I} = \emptyset$. Let P be a datalog$^\neg$ program. The immediate consequence operator T_P of P is extended to operate on sets of (positive and negative) ground literals as follows. Let $\mathbf{I}$ be a set of ground literals. $T_P(\mathbf{I})$ consists of all literals A for which there is a ground rule of P, $A \leftarrow L_1, \ldots, L_k$, such that $L_i \in \mathbf{I}$ for each i. Note that T_P can produce an inconsistent set of literals, which therefore does not correspond to a 3-valued model. Now let $\mathbf{I}$ be a set of ground literals and $\mathbf{J}$ a set of positive ground literals. $\mathbf{J}$ is said to be *an unfounded set* of P with respect to $\mathbf{I}$ if for each $A \in \mathbf{J}$ and ground rule r of P with A in the head, at least one of the following holds:

- the complement of some literal in the body of r is in $\mathbf{I}$; or
- some positive literal in the body of r is in $\mathbf{J}$.

Intuitively, this means that if all atoms of $\mathbf{I}$ are assumed true and all atoms in $\mathbf{J}$ are assumed false, then no atom of $\mathbf{J}$ is true under one application of T_P.

Let the *greatest unfounded set of P with respect to* $\mathbf{I}$ be the union of all unfounded sets of P with respect to $\mathbf{I}$, denoted $U_P(\mathbf{I})$. Next consider the operator W_P on sets of ground literals defined by

$$W_P(\mathbf{I}) = T_P(\mathbf{I}) \cup \neg.U_P(\mathbf{I}).$$

Prove the following:

(a) The greatest unfounded set $U_P(\mathbf{I})$ of P with respect to $\mathbf{I}$ is an unfounded set.

(b) The operator W_P is monotonic (with respect to set inclusion).

(c) The least fixpoint of W_P is consistent.

(d) The least fixpoint of W_P equals P^{wf}.

♠ **Exercise 15.25** [VanG89] Let P be a datalog$^\neg$ program. If $\mathbf{I}$ is a set of ground literals, let $P(\mathbf{I}) = T_P^\omega(\mathbf{I})$, where T_P is the immediate consequence operator on sets of ground literals defined in Exercise 15.24. Furthermore, $\overline{P}(\mathbf{I})$ denotes the complement of $P(\mathbf{I})$ [i.e., $\mathbf{B}(P, \mathbf{I}) - P(\mathbf{I})$]. Consider the sequence of sets of negative facts defined by

$$\begin{aligned} \mathbf{N}_0 &= \emptyset, \\ \mathbf{N}_{i+1} &= \neg.\overline{P}(\neg.\overline{P}(\mathbf{N}_i)). \end{aligned}$$

The intuition behind the definition is the following. $\mathbf{N}_0$ is an underestimate of the set of negative facts in the well-founded model. Then $P(\mathbf{N})$ is an underestimate of the positive facts, and the negated complement $\neg.\overline{P}(\mathbf{N})$ is an overestimate of the negative facts. Using this overestimate, one can infer an overestimate of the positive facts, $P(\neg.\overline{P}(\mathbf{N}))$. Therefore $\neg.\overline{P}(\neg.\overline{P}(\mathbf{N}))$ is now a new underestimate of the negative facts containing the previous underestimate. So $\{\mathbf{N}_i\}_{i \geq 0}$ is

an increasing sequence of underestimates of the negative facts, which converges to the negative facts in the well-founded model. Formally prove the following:

(a) The sequence $\{\mathbf{N}_i\}_{i\geq 0}$ is increasing.

(b) Let $\mathbf{N}$ be the limit of the sequence $\{\mathbf{N}_i\}_{i\geq 0}$ and $\mathbf{K} = \mathbf{N} \cup P(\mathbf{N})$. Then $\mathbf{K} = P^{wf}$.

(c) Explain the connection between the sequence $\{\mathbf{N}_i\}_{i\geq 0}$ and the sets of negative facts in the sequence $\{\mathbf{I}_i\}_{i\geq 0}$ defined in the alternating fixpoint computation of P^{wf} in the text.

(d) Suppose the definition of the sequence $\{\mathbf{N}_i\}_{i\geq 0}$ is modified such that $\mathbf{N}_0 = \neg.\mathbf{B}(P)$ (i.e., all facts are negative at the start). Show that for each $i \geq 0$, $\mathbf{N}_i = \neg.(\mathbf{I}_{2i})^0$.

Exercise 15.26 Let P be a datalog$^\neg$ program. Let T_P be the immediate consequence operator on sets of ground literals, defined in Exercise 15.24, and let $\bar{T}_P$ be defined by $\bar{T}_P(\mathbf{I}) = \mathbf{I} \cup T_P(\mathbf{I})$. Given a set $\mathbf{I}$ of ground literals, let $P(\mathbf{I})$ denote the limit of the increasing sequence $\{\bar{T}_P^i(\mathbf{I})\}_{i>0}$. A set $\mathbf{I}^-$ of negative ground literals is *consistent with respect to* P if $P(\mathbf{I}^-)$ is consistent. $\mathbf{I}^-$ is *maximally consistent with respect to* P if it is maximal among the sets of negative literals consistent with P. Investigate the connection between maximal consistency, 3-stable models, and well-founded semantics:

(a) Is $\neg.\mathbf{I}^0$ maximally consistent for every 3-stable model $\mathbf{I}$ of P?

(b) Is $P(\mathbf{I}^-)$ a 3-stable model of P for every $\mathbf{I}^-$ that is maximally consistent with respect to P?

(c) Is $\neg.(P^{wf})^0$ the intersection of all sets $\mathbf{I}^-$ that are maximally consistent with respect to P?

Exercise 15.27 Refer to the proof of Lemma 15.4.4.

(a) Outline a proof that $conseq_P$ can be simulated by a *while*$^+$ program.

(b) Provide a full description of the timestamping technique outlined in the proof of Lemma 15.4.4.

Exercise 15.28 Show that every query definable by stratified datalog$^\neg$ is a *fixpoint* query.

Exercise 15.29 Consider an ordered database (i.e., with binary relation *succ* providing a successor relation on the constants). Prove that the minimum and maximum constants cannot be computed using a semipositive program.

★ **Exercise 15.30** Consider the game trees and *winning* query described in Section 15.4.

(a) Show that *winning* is true on the game trees $G_{2i,k}$ and false on the game trees $G'_{2i,k}$, for $i > 0$.

(b) Prove that the *winning* query on game trees is defined by the *fixpoint* query exhibited in Section 15.4.