```
for each predicate p do
    stratum[p] := 1;
repeat
    for each rule r with head predicate p do begin
        for each negated subgoal of r with predicate q do
            stratum[p] := max(stratum[p], 1+stratum[q]);
        for each nonnegated subgoal of r with predicate q do
            stratum[p] := max(stratum[p], stratum[q])
    end
until there are no changes to any stratum
    or some stratum exceeds the number of predicates
```

Figure 3.7 Stratification computation.

Example 3.19: If there are no negated subgoals, then Algorithm 3.5 immediately halts with all predicates in stratum 1, which is correct.

For a less trivial example, consider the rules of Example 3.17. Initially, p, q, and r each have stratum 1. Rule (1) forces us to increase the stratum of p to 2, and then the rule (2) forces us to increase the stratum of q to 3. The first rule then requires the stratum of p to be 4. We now have a stratum higher than the number of predicates, and so conclude the rules are not stratifiable. That conclusion is correct. For example, q appears as a negated subgoal in the first rule, which has head predicate p, yet q depends on p.

For another example, consider the rules of Example 3.18. Starting with all four predicates in stratum 1, rule (3) forces us to increase q to stratum 2. However, there are no further adjustments that need to be made. We conclude p, r, and s are in stratum 1; q is in stratum 2. That makes sense, because r and s are EDB relations, while p is an IDB relation that can be computed from these by Algorithm 3.3 or 3.4, without any uses of negation. After we have computed p, we can then pretend the negation of p is an EDB relation. Since q is not recursive, we can compute the relation for q by Algorithm 3.2. □

The correctness of Algorithm 3.5 is proved by the following lemmas and theorem.

Lemma 3.2: If a logic program has a stratification, then it is stratified.

Proof: The reader should not be lulled by the similarity of the terms "stratified" and "has a stratification" into thinking that they are easy to prove equivalent. In fact they are equivalent, but the proof requires some work. Recall that a program is stratified if whenever there is a negated subgoal with predicate q in the body of a rule for predicate p, there is no path in the dependency graph from p to q. It is easy to see from the definition of a "stratification" that the stratum of predicates along any path in the dependency graph can never decrease, because those paths go from subgoal to head, and the stratum of a head predicate is never less than the stratum of one of its subgoals.

Suppose a program had a stratification, but was not stratified. Then there would be a path in the dependency graph to some q from some p, such that negated q was a subgoal of a rule r for p. The existence of the path says that the stratum of q is at least as high as the stratum of p, yet the rule r requires that the stratum of q be less than that of p. □

Lemma 3.3: If a logic program is stratified, then Algorithm 3.5 halts on that program without producing a stratum higher than n, the number of predicates in the program.

Proof: Each time we increase the stratum of some predicate p because of some predicate q in the algorithm of Figure 3.7, it must be that q is a subgoal (negated or not) of a rule for p. If we increase *stratum*[p] to i, and q is not negated, then write $q \xrightarrow{i} p$; if q is negated, write $q \overset{i}{\Rightarrow} p$. For example, the sequence of stratum changes discussed in Example 3.19 for the nonstratified rules of Example 3.17 is $q \overset{2}{\Rightarrow} p \overset{3}{\Rightarrow} q \overset{4}{\Rightarrow} p$.

For technical reasons, it is convenient to add a new symbol *start*, which is assumed not to be a predicate. We then let $start \overset{1}{\Rightarrow} p$ for all predicates p.

It is an easy induction on the number of times Algorithm 3.5 changes a stratum that if we set the stratum of a predicate p to i, then there is a chain of $\rightarrow$ and $\Rightarrow$ steps from *start* to p that includes at least $i \Rightarrow$ steps. The key point in the proof is that if the last step by which Algorithm 3.5 makes the stratum of p reach i is $q \overset{i}{\Rightarrow} p$, then there is a chain with at lease $i-1 \Rightarrow$ steps to q, and one more makes at least $i \Rightarrow$'s to p. If the step by which Algorithm 3.5 makes the stratum of p reach i is $q \xrightarrow{i} p$, then there is already a chain including at least $i \Rightarrow$'s to q, and this chain can be extended to p.

Now, notice that if the stratum of some predicate reaches $n+1$, there is a chain with at least $n+1 \Rightarrow$'s. Thus some predicate, say p, appears twice as the head of a $\Rightarrow$. Thus, a part of the chain is

$$q_1 \overset{i}{\Rightarrow} p \cdots q_2 \overset{j}{\Rightarrow} p$$

where $i < j$. Also, observe that every portion of the chain is a path in the dependency graph; in particular, there is a path from p to q_2 in the dependency graph.

The fact that $q_2 \overset{j}{\Rightarrow} p$ is a step implies that there is a rule with head p and negated subgoal q_2. Thus, there is a path in the dependency graph from the head, p, of some rule to a negated subgoal, q_2, of that rule, contradicting the assumption that the logic program is stratified. We conclude that if the program is stratified, no stratum produced by Algorithm 3.5 ever exceeds n, and therefore, Algorithm 3.5 must eventually halt and answer "yes." □

Theorem 3.6: Algorithm 3.5 correctly determines whether a datalog program with negation is stratified.

Proof: Evidently, if Algorithm 3.5 halts and says the program is stratified, then it has produced a valid stratification. Lemma 3.2 says that if there is a stratification, then the program is stratified, and Lemma 3.3 says that if the logic program is stratified, then Algorithm 3.5 halts and says "yes" (the program is stratified). We conclude that the algorithm says "yes" if and only if the given logic program is stratified. □

Corollary 3.3: A logic program is stratified if and only if it has a stratification.

Proof: The three-step implication in the proof of Theorem 3.6 incidentally proves that the three conditions "stratified," "has a stratification," and "Algorithm 3.5 says 'yes'," all are equivalent. □

Safe, Stratified Rules

In order that a sensible meaning for rules can be defined we need more than stratification; we need safety. Recall that we defined rules to be "safe" in Section 3.2 if all their variables were limited, either by being an argument of a nonnegated, ordinary subgoal, or by being equated to a constant or to a limited variable, perhaps through a chain of equalities. When we have negated subgoals, the definition of "safe" does not change. We are not allowed to use negated subgoals to help prove variables to be limited.

Example 3.20: The rules of Examples 3.16, 3.17, and 3.18 are all safe. The rule of Example 3.15 is not safe, since Y appeared in a negated subgoal but in no nonnegated subgoal, and therefore could not be limited. However, as we saw in that example, we can convert that rule to a pair of safe rules that intuitively mean the same thing. □

When rules are both safe and stratified, there is a natural choice from among possible fixed points that we shall regard as the "meaning" of the rules. We process each stratum in order, starting with the lowest first. Suppose we are working on a predicate p of stratum i. If a rule for p has a subgoal with a predicate q of stratum less than i, we can obtain q's relation, because that relation is either an EDB relation or has been computed when we worked on previous strata. Of course, no subgoal can be of stratum above i, if we have a valid stratification. Moreover, if the subgoal is negated, then stratum of q must be strictly less than i.

As a consequence of these properties of a stratification, we can view the set of rules for the predicates of stratum i as a recursive definition of the relations for exactly the stratum-i predicates, in terms of relations for the EDB relations and all IDB relations of lower strata. As the equations for the IDB predicates

of stratum i have no negated subgoals of stratum i, we may apply Algorithm 3.3 or 3.4 to solve them.

The only technicality concerns how we create the relation for a negated subgoal $\neg q(X_1, \ldots, X_n)$ of a rule r, so that we may pretend it is the finite relation belonging to some nonnegated subgoal. Define DOM to be the union of the symbols appearing in the EDB relations and in the rules themselves. As we argued, in safe rules, no symbol not in the EDB or the rules can appear in a substitution that makes the body of a rule true. Therefore, we lose nothing by restricting the relation for a negated subgoal to consist only of tuples whose values are chosen from DOM.

Thus, let Q be the relation already computed for q (or given, if q is an EDB predicate). Let the relation $\bar{Q}$ for subgoal $\neg q(X_1, \ldots, X_n)$ be

$$DOM \times \cdots \times DOM \ (n \text{ times}) - Q$$

If we make the analogous substitution for each negated subgoal in the rules for stratum i, and then apply Algorithm 3.3 to compute the least fixed point for the IDB predicates of that one stratum, i, then we shall obtain the same result as if we had managed to use the infinite relation of all tuples not in Q in place of $\bar{Q}$. The reason is that we can prove, in an easy induction on the number of rounds of Algorithm 3.3, that every tuple we add to an IDB relation of stratum i consists of tuples whose components are all in DOM. The proof depends only on the observation that, as the rules are safe, every variable appearing in a negated subgoal appears also in a nonnegated subgoal. Given that, we can invoke the inductive hypothesis to show that at each round the relation for any rule will be a subset of $DOM \times \cdots \times DOM$. We can thus offer the following algorithm for computing the "perfect" fixed point of a datalog program with safe, stratified negation.

Algorithm 3.6: Evaluation of Relations for Safe, Stratified Datalog Programs.

INPUT: A datalog program whose rules are safe, rectified, and stratified. Also, relations for all the EDB predicates of the program.

OUTPUT: Relations for all the IDB predicates, forming a minimal fixed point of the datalog program.

METHOD: First, compute the stratification for the program by Algorithm 3.5. Compute DOM by projecting all EDB relations onto each of their components and then taking the union of these projections and the set of constants appearing in the rules, if any.

Then for each stratum i, in turn, do the following steps. When we reach stratum i, we have already computed the relations for the IDB predicates at lower strata, and of course we are given the relations for the EDB predicates. Thus, in particular, if a rule at stratum i has a negated subgoal, the relation for that subgoal is known.

1. Consider each nonnegated subgoal in a rule for stratum i. If that subgoal is an EDB predicate or an IDB predicate at a stratum below i, use the relation already known for that predicate.
2. For each negated subgoal in a rule for stratum i, let Q be the relation for its predicate; Q must have been computed, because the stratum of the predicate for the negated subgoal is less than i. If this subgoal has arity n, use the relation $DOM \times \cdots \times DOM - Q$ in place of this subgoal, where DOM appears n times in the product.
3. Use Algorithm 3.3 or 3.4 to compute the relations for the IDB predicates of stratum i, treating all the subgoals whose relations were obtained in either step (2) or step (3), as if they were EDB relations with the values given by those steps. □

Example 3.21: Consider the rules of Example 3.18. Recall from Example 3.19 that p, r, and s are in stratum 1, and q is in stratum 2. The relations R and S for r and s are given EDB relations. Since all EDB relations are of arity 1, and there are no constants in the rules, DOM is just $\pi_1(R) \cup \pi_1(S)$, or $R \cup S$.

We first work on stratum 1, and we merely need one round to find that P, the relation for p is equal to R. Now, we proceed to stratum 2. Since p appears negated, we must compute $\bar{P}$, a relation that includes all tuples that could possibly be in the relation Q for q, yet are not in P. This relation is $DOM - P = R \cup S - P$. Since $R = P$ was just established, $\bar{P} = S - R$ here. Since there is no recursion in stratum 2, we immediately get

$$Q(X) = S(X) \bowtie \bar{P}(X) = S(X) \cap \bar{P}(X) = \\ S(X) \cap \big(S(X) - R(X)\big) = S(X) - R(X)$$

That is, $Q(X) = S(X) - R(X)$.[17]

In Example 3.18, we observed that there could be more than one minimal fixed point. The fixed point produced by Algorithm 3.6 corresponds to the fixed point $\mathcal{S}_1$ of that example. □

Perfect Fixed Points

Let us call the fixed point computed by Algorithm 3.6 the *perfect* fixed point or model. There is little we can prove about Algorithm 3.6, since as we understand from Example 3.18, it simply computes one of a number of minimal fixed points for a set of safe, stratified rules with negation. Technically, we never proved that what Algorithm 3.6 produces *is* a minimal fixed point. However, the fact that we have a fixed point follows from Theorem 3.4 (the correctness of Algorithm

[17] The reader should not assume from this example that the result of Algorithm 3.6 is always a formula in relational algebra for the IDB relations. Generally, when the rules are recursive, there is no such formula, and the only reason we have one in this case is that the recursion of rule (2) in Example 3.18 is trivial.

3.3 for computing least fixed points when there is no negation) and a simple induction on the strata. That is, we show by induction on i that the equations derived from the rules with heads of stratum i are satisfied.

As for showing we have a minimal fixed point, we can actually show more. The perfect fixed point $\mathcal{S}$ has the following properties:

1. If $\mathcal{S}_1$ is any other fixed point, then for every predicate p of stratum 1, p's relation in $\mathcal{S}$ is a subset (not necessarily proper) of p's relation in $\mathcal{S}_1$.
2. For all $i > 1$, if $\mathcal{S}_1$ is any fixed point that agrees with $\mathcal{S}$ on the relations for all predicates of strata less than i, then the relations for the predicates of stratum i are subsets in $\mathcal{S}$ of their relations in $\mathcal{S}_1$.

It follows from (1) and (2) that $\mathcal{S}$ is a minimal fixed point. In fact, $\mathcal{S}$ is "least" of all minimal fixed points if one puts the most weight on having small relations at the lowest strata. All the results mentioned above are easy inductions on the strata, and we shall leave them as exercises for the reader.

3.7 RELATIONAL ALGEBRA AND LOGIC

We can view relational algebra expressions as defining functions that take given relations as arguments and that produce a value, which is a computed relation. Likewise, we know that datalog programs take EDB relations as arguments and produce IDB relations as values. We might ask whether the functions defined by relational algebra and by logic programs are the same, or whether one notation is more expressive than the other.

The answer, as we shall prove in this section, is that without negation in rules, relational algebra and datalog are incommensurate in their expressive power; there are things each can express that the other cannot. With negation, datalog is strictly more expressive than relational algebra. In fact, the set of functions expressible in relational algebra is equivalent to the set of functions we can express in datalog (with negation) if rules are restricted to be safe, nonrecursive, and have only stratified negation. In this section, "nonrecursive datalog" will be assumed to refer to rules of this form unless stated otherwise. Note that since the rules are nonrecursive, it is easy to see that they must be stratified.

From Relational Algebra to Logical Rules

Mimicking the operations of relational algebra with datalog rules is easy except for selections that involve complex conditions. Thus, we begin with two lemmas that let us break up selections by arbitrary formulas into a cascade of unions and selections by simpler formulas. Then we give a construction of rules from arbitrary relational algebra formulas.

Lemma 3.4: Every selection is equivalent to a selection that does not use the NOT operator.